Kanupriya Gulati
Sunil P. Khatri

Hardware Acceleration of EDA Algorithms

Custom ICs, FPGAs and GPUs

Springer

Hardware Acceleration of EDA Algorithms

Kanupriya Gulati · Sunil P. Khatri

Hardware Acceleration of EDA Algorithms

Custom ICs, FPGAs and GPUs

Springer

Kanupriya Gulati
109 Branchwood Trl
Coppell TX 75019
USA
kgulati@tamu.edu

Sunil P. Khatri
Department of Electrical & Computer Engineering
Texas A & M University
College Station TX
77843-3128
214 Zachry Engineering Center
USA
sunilkhatri@tamu.edu

ISBN 978-1-4419-0943-5 e-ISBN 978-1-4419-0944-2
DOI 10.1007/978-1-4419-0944-2
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010920238

© Springer Science+Business Media, LLC 2010
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To our parents and our teachers

Foreword

Single-threaded software applications have ceased to see significant gains in performance on a general-purpose CPU, even with further scaling in very large scale integration (VLSI) technology. This is a significant problem for electronic design automation (EDA) applications, since the design complexity of VLSI integrated circuits (ICs) is continuously growing. In this research monograph, we evaluate custom ICs, field-programmable gate arrays (FPGAs), and graphics processors as platforms for accelerating EDA algorithms, instead of the general-purpose single-threaded CPU. We study applications which are used in key time-consuming steps of the VLSI design flow. Further, these applications also have different degrees of inherent parallelism in them. We study both control-dominated EDA applications and control plus data parallel EDA applications. We accelerate these applications on these different hardware platforms. We also present an automated approach for accelerating certain uniprocessor applications on a graphics processor.

This monograph compares custom ICs, FPGAs, and graphics processing units (GPUs) as potential platforms to accelerate EDA algorithms. It also provides details of the programming model used for interfacing with the GPUs. As an example of a control-dominated EDA problem, Boolean satisfiability (SAT) is accelerated using the following hardware implementations: (i) a custom IC-based hardware approach in which the traversal of the implication graph and conflict clause generation are performed in hardware, in parallel, (ii) an FPGA-based hardware approach to accelerate SAT in which the entire SAT search algorithm is implemented in the FPGA, and (iii) a complete SAT approach which employs a new GPU-enhanced variable ordering heuristic.

In this monograph, several EDA problems with varying degrees of control and data parallelisms are accelerated using a general-purpose graphics processor. In particular we accelerate Monte Carlo based statistical static timing analysis, device model evaluation (for accelerating circuit simulation), fault simulation, and fault table generation on a graphics processor, with speedups of up to 800×. Additionally, an automated approach is presented that accelerates (on a graphics processor) uniprocessor code that is executed multiple times on independent data sets in an application. The key idea here is to partition the software into kernels in an automated fashion, such that multiple independent instances of these kernels, when

executed in parallel on the GPU, can maximally benefit from the GPU's hardware resources.

We hope that this monograph can serve as a valuable reference to individuals interested in exploring alternative hardware platforms and to those interested in accelerating various EDA applications by harnessing the parallelism in these platforms.

College Station, TX — Kanupriya Gulati
College Station, TX — Sunil P. Khatri
October 2009

Preface

In recent times, serial software applications have no longer enjoyed significant gains in performance with process scaling, since microprocessor performance gains have been hampered due to increases in power and manufacturability issues, which accompany scaling. With the continuous growth of IC design complexities, this problem is particularly significant for EDA applications. In this research monograph, we evaluate the feasibility of hardware platforms such as custom ICs, FPGAs, and graphics processors, for accelerating EDA algorithms. We choose applications which contribute significantly to the total runtime of the VLSI design flow and which have varied degrees of inherent parallelism in them. We study the acceleration of such algorithms on these alternative platforms. We also present an automated approach to accelerate certain specific types of uniprocessor subroutines on the GPU.

This research monograph consists of four parts. The alternative hardware platforms, along with the details of the programming model used for interfacing with the graphics processing units, are discussed in the first part of this monograph. The second part of this monograph studies the acceleration of an algorithm in the *control-dominated* category, namely Boolean satisfiability (SAT). The third part studies the acceleration of some algorithms in the *control plus data parallel* category, namely Monte Carlo based statistical static timing analysis, circuit simulation, fault simulation and fault table generation. In the fourth part of the monograph, we present the automated approach to generate GPU code to accelerate certain software subroutines.

Book Outline

This research monograph is organized into four parts. In Part I of this research monograph, we discuss alternative hardware platforms. We also provide details of the programming model used for interfacing with the graphics processor. In Chapter 2, we compare and contrast the hardware platforms that are considered in this monograph. In particular, we discuss custom-designed ICs, reconfigurable architectures such as FPGAs, and streaming processors such as graphics processing units

(GPUs). This comparison is performed over various criteria such as architecture, expected performance, programming model and environment, scalability, time to market, security, and cost of hardware. In Chapter 3, we describe the programming environment used for interfacing with the GPUs.

In Part II of this monograph we present hardware implementations of a control-dominated EDA problem, namely Boolean satisfiability (SAT). We present approaches to accelerate SAT using each of the three hardware platforms under consideration. In Chapter 4, we present a custom IC-based hardware approach to accelerate SAT. In this approach, the traversal of the implication graph and conflict clause generation are performed in hardware, in parallel. Further, we propose a hardware approach to extract the minimum unsatisfiable core for any unsatisfiable formula. In Chapter 5, we discuss an FPGA-based hardware approach to accelerate SAT. In this approach, we store the clauses in the FPGA slices. In order to solve large SAT instances, we partition the instance into 'bins,' each of which can fit in the FPGA. The solution of SAT clauses of any bin is performed in parallel. Our approach also handles (in hardware) the fact that the original SAT instance is partitioned into bins. In Chapter 6, we present a SAT approach which employs a new GPU-enhanced variable ordering heuristic. In this approach, we augment a CPU-based complete procedure (MiniSAT), with a GPU-based approximate procedure (survey propagation). In this manner, the complete procedure benefits from the high parallelism of the GPU.

In Part III of this book, we study the acceleration of several EDA problems, with varying amounts of control and data parallelism, on a GPU. In Chapter 7, we exploit the parallelism in Monte Carlo based statistical static timing analysis and accelerate it on a graphics processor. In this approach, we map the Monte Carlo based SSTA computations to the large number of threads that can be computed in parallel on a GPU. Our approach performs multiple delay simulations of a single gate in parallel and further benefits from a parallel implementation of the Mersenne Twister pseudo-random number generator on the GPU, followed by Box–Muller transformations (also implemented on the GPU). In Chapter 8, we study the acceleration of fault simulation on a GPU. Fault simulation is inherently parallelizable and requires a large number of gate evaluations to be performed for each gate in a design. The large number of threads that can be computed in parallel on a GPU can be employed to perform a large number of these gate evaluations in parallel. We implement a pattern and fault parallel fault simulator, which fault-simulates a circuit in a levelized fashion. We ensure that all threads of the GPU compute identical instructions, but on different data. We study the generation of a fault table using a GPU in Chapter 9. We employ a pattern parallel approach, which utilizes both bit parallelism and thread-level parallelism. In Chapter 10, we explore the GPU-based acceleration of the model card evaluation of a circuit simulator. Our resulting code is integrated into a commercial fast SPICE tool, and the overall speedup obtained is measured. With careful engineering, we maximally harness the GPU's immense memory bandwidth and high computational power.

In Part IV of this book, we present an automated approach to accelerate uniprocessor subroutines which are required to be executed multiple times within an

application, on independent data sets. The target hardware platform is a general-purpose graphics platform. The key idea here is to partition the subroutine into kernels in an automated fashion, such that multiple instances of these kernels, when executed in parallel on the GPU, can maximally benefit from the GPU's hardware resources. This approach is detailed in Chapter 11.

The approaches presented in this monograph collectively aim to contribute toward enabling the VLSI CAD community to accelerate EDA algorithms on different hardware platforms.

College Station, TX — Kanupriya Gulati
College Station, TX — Sunil P. Khatri
October 2009

Acknowledgments

The work presented in this research monograph would not have been possible without the tremendous amount of help and encouragement we have received from our families, friends, and colleagues.

In particular, we are grateful to Mandar Waghmode, who contributed toward the custom IC-based engine for accelerating Boolean satisfiability; Dr. Srinivas Patil, Dr. Abhijit Jas, and Suganth Paul, for their assistance on the FPGA-based approach for accelerating Boolean satisfiability; and Dr. John Croix and Rahm Shastry, who helped in integrating our GPU-based accelerated code for model card evaluation into a commercial fast SPICE tool.

We acknowledge the insightful comments of Dr. Peng Li, Dr. Hank Walker, Dr. Desmond Kirkpatrick, and Dr. Jim Ji. We would also like to thank Intel Corporation, Nascentric Inc., Accelicon Technologies Inc., and NVIDIA Corporation, for supporting this research through research grants and an NVIDIA fellowship, respectively.

Contents

1 Introduction 1
1.1 Hardware Platforms Considered in This Research Monograph 3
1.2 EDA Algorithms Studied in This Research Monograph 3
1.2.1 Control-Dominated Applications 4
1.2.2 Control Plus Data Parallel Applications 4
1.3 Automated Approach for GPU-Based Software Acceleration 4
1.4 Chapter Summary 4
References 5

Part I Alternative Hardware Platforms

2 Hardware Platforms 9
2.1 Chapter Overview 9
2.2 Introduction 9
2.3 Hardware Platforms Studied in This Research Monograph 10
2.3.1 Custom ICs 10
2.3.2 FPGAs 10
2.3.3 Graphics Processors 10
2.4 General Overview and Architecture 11
2.5 Programming Model and Environment 14
2.6 Scalability 15
2.7 Design Turn-Around Time 16
2.8 Performance 16
2.9 Cost of Hardware 18
2.10 Floating Point Operations 18
2.11 Security and Real-Time Applications 19
2.12 Applications 19
2.13 Chapter Summary 20
References 20

3 GPU Architecture and the CUDA Programming Model 23
3.1 Chapter Overview 23
3.2 Introduction 23
3.3 Hardware Model 24
3.4 Memory Model 25
3.5 Programming Model 28
3.6 Chapter Summary 30
References 30

Part II Control-Dominated Category

4 Accelerating Boolean Satisfiability on a Custom IC 33
4.1 Chapter Overview 33
4.2 Introduction 34
4.3 Previous Work 36
4.4 Hardware Architecture 37
4.4.1 Abstract Overview 37
4.4.2 Hardware Overview 38
4.4.3 Hardware Details 39
4.5 An Example of Conflict Clause Generation 50
4.6 Partitioning the CNF Instance 51
4.7 Extraction of the Unsatisfiable Core 53
4.8 Experimental Results 54
4.9 Chapter Summary 59
References 59

5 Accelerating Boolean Satisfiability on an FPGA 63
5.1 Chapter Overview 63
5.2 Introduction 64
5.3 Previous Work 64
5.4 Hardware Architecture 66
5.4.1 Architecture Overview 66
5.5 Solving a CNF Instance Which Is Partitioned into Several Bins . . . 67
5.6 Partitioning the CNF Instance 69
5.7 Hardware Details 70
5.8 Experimental Results 72
5.8.1 Current Implementation 72
5.8.2 Performance Model 73
5.8.3 Projections 77
5.9 Chapter Summary 80
References 80

6 Accelerating Boolean Satisfiability on a Graphics Processing Unit ... 83
6.1 Chapter Overview ... 83
6.2 Introduction ... 83
6.3 Related Previous Work ... 85
6.4 Our Approach ... 87
6.4.1 SurveySAT and the GPU ... 87
6.4.2 MiniSAT Enhanced with Survey Propagation (MESP) ... 93
6.5 Experimental Results ... 96
6.6 Chapter Summary ... 98
References ... 98

Part III Control Plus Data Parallel Applications

7 Accelerating Statistical Static Timing Analysis Using Graphics Processors ... 105
7.1 Chapter Overview ... 105
7.2 Introduction ... 106
7.3 Previous Work ... 108
7.4 Our Approach ... 109
7.4.1 Static Timing Analysis (STA) at a Gate ... 109
7.4.2 Statistical Static Timing Analysis (SSTA) at a Gate ... 112
7.5 Experimental Results ... 113
7.6 Chapter Summary ... 116
References ... 116

8 Accelerating Fault Simulation Using Graphics Processors ... 119
8.1 Chapter Overview ... 119
8.2 Introduction ... 119
8.3 Previous Work ... 121
8.4 Our Approach ... 122
8.4.1 Logic Simulation at a Gate ... 123
8.4.2 Fault Injection at a Gate ... 125
8.4.3 Fault Detection at a Gate ... 126
8.4.4 Fault Simulation of a Circuit ... 127
8.5 Experimental Results ... 129
8.6 Chapter Summary ... 131
References ... 131

9 Fault Table Generation Using Graphics Processors ... 133
9.1 Chapter Overview ... 133
9.2 Introduction ... 134
9.3 Previous Work ... 136
9.4 Our Approach ... 136

9.4.1 Definitions . . . 137
9.4.2 Algorithms: FSIM∗ and GFTABLE . . . 139
9.5 Experimental Results . . . 146
9.6 Chapter Summary . . . 150
References . . . 151

10 Accelerating Circuit Simulation Using Graphics Processors . . . 153
10.1 Chapter Overview . . . 153
10.2 Introduction . . . 153
10.3 Previous Work . . . 155
10.4 Our Approach . . . 157
10.4.1 Parallelizing BSIM3 Model Computations on a GPU . . . 158
10.5 Experiments . . . 162
10.6 Chapter Summary . . . 165
References . . . 165

Part IV Automated Generation of GPU Code

11 Automated Approach for Graphics Processor Based Software Acceleration . . . 169
11.1 Chapter Overview . . . 169
11.2 Introduction . . . 169
11.3 Our Approach . . . 171
11.3.1 Problem Definition . . . 171
11.3.2 GPU Constraints on the Kernel Generation Engine . . . 172
11.3.3 Automatic Kernel Generation Engine . . . 173
11.4 Experimental Results . . . 176
11.4.1 Evaluation Methodology . . . 177
11.5 Chapter Summary . . . 179
References . . . 179

12 Conclusions . . . 181
References . . . 187

Index . . . 189

List of Tables

4.1 Encoding of {*reg*,*reg_bar*} bits 41
4.2 Encoding of {*lit*,*lit_bar*} and *var_implied* signals 42
4.3 Partitioning and binning results 55
4.4 Comparing against MiniSAT (a BCP-based software SAT solver) 57
5.1 Number of bins touched with respect to bin size 76
5.2 LUT distribution for FPGA devices 76
5.3 Runtime comparison XC4VFX140 versus MiniSAT 79
6.1 Comparing MiniSAT with SurveySAT (CPU) and SurveySAT (GPU) . . 94
6.2 Comparing MESP with MiniSAT 97
7.1 Monte Carlo based SSTA results 115
8.1 Encoding of the mask bits 126
8.2 Parallel fault simulation results 130
9.1 Fault table generation results with $L = 32K$ 148
9.2 Fault table generation results with $L = 8K$ 149
9.3 Fault table generation results with $L = 16K$ 150
10.1 Speedup for BSIM3 evaluation 162
10.2 Speedup for circuit simulation 163
11.1 Validation of the automatic kernel generation approach 178

List of Figures

1.1 CPU performance growth [3] . . . 2
2.1 FPGA layout [14] . . . 12
2.2 Logic block in the FPGA . . . 12
2.3 LUT implementation using a 16:1 MUX . . . 13
2.4 SRAM configuration bit design . . . 13
2.5 Comparing Gflops of GPUs and CPUs [11] . . . 14
2.6 FPGA growth trend [9] . . . 17
3.1 CUDA for interfacing with GPU device . . . 24
3.2 Hardware model of the NVIDIA GeForce GTX 280 . . . 25
3.3 Memory model of the NVIDIA GeForce GTX 280 . . . 26
3.4 Programming model of CUDA . . . 28
4.1 Abstracted view of the proposed idea . . . 37
4.2 Generic floorplan . . . 38
4.3 State diagram of the decision engine . . . 39
4.4 Signal interface of the clause cell . . . 40
4.5 Schematic of the clause cell . . . 41
4.6 Layout of the clause cell . . . 43
4.7 Signal interface of the base cell . . . 43
4.8 Indicating a new implication . . . 44
4.9 Computing backtrack level . . . 46
4.10 **(a)** Internal structure of a bank. **(b)** Multiple clauses packed in one bank-row . . . 47
4.11 Signal interface of the terminal cell . . . 47
4.12 Schematic of a terminal cell . . . 48
4.13 Hierarchical structure for inter-bank communication . . . 49
4.14 Example of implicit traversal of implication graph . . . 51
5.1 Hardware architecture . . . 67
5.2 State diagram of the decision engine . . . 71
5.3 Resource utilization for clauses . . . 73
5.4 Resource utilization for variables . . . 74
5.5 Computing aspect ratio (16 variables) . . . 75
5.6 Computing aspect ratio (36 variables) . . . 75
6.1 Data structure of the SAT instance on the GPU . . . 92

7.1 Comparing Monte Carlo based SSTA on GTX 280 GPU and Intel Core 2 processors (with SEE instructions) 116
8.1 Truth tables stored in a lookup table 123
8.2 Levelized logic netlist 128
9.1 Example circuit 137
9.2 CPT on FFR(k) 142
9.3 Fault simulation on SR(k) 145
10.1 Industrial_2 waveforms 164
10.2 Industrial_3 waveforms 164
11.1 CDFG example 174
11.2 KDG example 175
12.1 New parallel kernel GPUs 184
12.2 Larrabee architecture from Intel 185
12.3 Fermi architecture from NVIDIA 185
12.4 Block diagram of a single shared multiprocessor (SM) in Fermi 186
12.5 Block diagram of a single processor (core) in SM 187

Part I
Alternative Hardware Platforms

Outline of Part I

In this research monograph, we explore the following hardware platforms for accelerating EDA applications:

- Custom-designed ICs are arguably the fastest accelerators we have today, easily offering several orders of magnitude speedup compared to the single-threaded software performance on the CPU.These chips are application specific, and thus deliver high performance for the target application, albeit at a high cost.
- Field-programmable gate arrays (FPGAs) have been popular for hardware prototyping for several years now. Hardware designers have used FPGAs for implementing system-level logic including state machines, memory controllers, 'glue' logic, and bus interfaces. FPGAs have also been heavily used for system prototyping and for emulation purposes. More recently, high-performance systems have begun to increasingly utilize FPGAs. This has been made possible in part because of increased FPGA device densities, by advances in FPGA tool flows, and also by the increasing cost of application-specific integrated circuit (ASIC) or custom IC implementations.
- Graphics processing units (GPUs) are designed to operate in a single instruction multiple data (SIMD) fashion. The key application of a GPU is to serve as a graphics accelerator for speeding up image processing, 3D rendering operations, etc., as required of a graphics card in a CPU. In general, these graphics acceleration tasks perform the same operation (i.e., instructions) independently on large volumes of data. The application of GPUs for general-purpose computations has been actively explored in recent times. The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data-intensive algorithms has arguably had a contribution in encouraging GPU manufacturers to design easily programmable general-purpose GPUs (GPGPUs). GPU architectures have been continuously evolving toward higher performance, larger memory sizes, larger memory bandwidths, and relatively lower costs.

Part I of this monograph is organized as follows. The above-mentioned hardware platforms are compared and contrasted in Chapter 2, using criteria such as architecture, expected performance, programming model and environment, scalability, time to market, security, and cost of hardware. In Chapter 3, we describe the programming environment used for interfacing with the GPU devices.

Chapter 1
Introduction

With the advances in VLSI technology over the past few decades, several software applications got a 'free' performance boost, without needing any code redesign. The steadily increasing clock rates and higher memory bandwidths resulted in improved performance with zero software cost. However, more recently, the gain in the single-core performance of general-purpose processors has diminished due to the decreased rate of increase of operating frequencies. This is because VLSI system performance hit two big *walls*:

- the *memory wall* and
- the *power wall*.

The memory wall refers to the increasing gap between processor and memory speeds. This results in an increase in cache sizes required to hide memory access latencies. Eventually the memory bandwidth becomes the bottleneck in performance. The power wall refers to power supply limitations or thermal dissipation limitations (or both) – which impose a hard constraint on the total amount of power that processors can consume in a system. Together, these two walls reduce the performance gains expected for general-purpose processors, as shown in Fig. 1.1. Due to these two factors, the rate of increase of processor frequency has greatly decreased. Further, the VLSI system performance has not shown much gain from continued processor frequency increases as was once the case.

Further, newer manufacturing and device constraints are faced with decreasing feature sizes, making future performance increases harder to obtain. A leading processor design company summarized the causes of reduced speed improvements in their white paper [1], stating:

> First of all, as chip geometries shrink and clock frequencies rise, the transistor leakage current increases, leading to excess power consumption and heat ... Secondly, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies. Third, for certain applications, traditional serial architectures are becoming less efficient as processors get faster (due to the so-called Von Neumann bottleneck), further undercutting any gains that frequency increases might otherwise buy. In addition, partly due to limitations in the means of producing inductance within solid state devices, resistance-capacitance (RC) delays in signal transmission are growing as feature sizes shrink, imposing an additional bottleneck that frequency increases don't address.

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_1,
© Springer Science+Business Media, LLC 2010

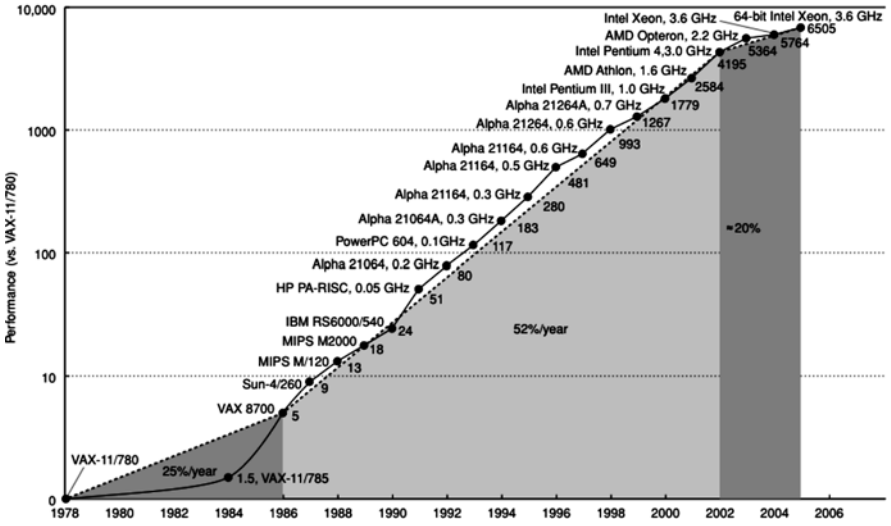


Fig. 1.1 CPU performance growth [3]

In order to maintain increasing peak performance trends without being hit by these 'walls,' the microprocessor industry rapidly shifted to multi-core processors. As a consequence of this shift in microprocessor design, traditional single-threaded applications no longer see significant gains in performance with each processor generation, unless these applications are rearchitectured to take advantage of the multi-core processors. This is due to the *instruction-level parallelism (ILP) wall*, which refers to the rising difficulty in finding enough parallelism in the existing instructions stream of a single process, making it hard to keep multiple cores busy. The ILP wall further compounds the difficulty of performance scaling at the application level. These *walls* are a key problem for several software applications, including software for electronic design.

The electronic design automation (EDA) field collectively uses a diverse set of software algorithms and tools, which are required to design complex next-generation electronics products. The increase in VLSI design complexity poses a challenge to the EDA community, since single-thread performance is not scaling effectively due to reasons mentioned above. Parallel hardware presents an opportunity to solve this dilemma and opens up new design automation opportunities which yield orders of magnitude faster algorithms. In addition to multi-core processors, other hardware platforms may be viable alternatives to achieve this acceleration as well. These include custom-designed ICs, reconfigurable hardware such as FPGAs, and streaming processors such as graphics processing units. All these alternatives need to be investigated as potential solutions for accelerating EDA applications. This research monograph studies the feasibility of using these alternative platforms for a subset of EDA applications which

- address some extremely important steps in the VLSI design flow and
- have varying degrees of inherent parallelism in them.

The rest of this chapter is organized as follows. In the next section, we briefly introduce the hardware platforms that are studied in this monograph. In Section 1.2 we discuss the EDA applications considered in this monograph. In Section 1.3 we discuss our approach to automatically generate graphics processing unit (GPU) based code to accelerate uniprocessor software. Section 1.4 summarizes this chapter.

1.1 Hardware Platforms Considered in This Research Monograph

In this book, we explore the three following hardware platforms for accelerating EDA applications. *Custom-designed ICs* are arguably the fastest accelerators we have today, easily offering several orders of magnitude speedup compared to the single-threaded software performance on the CPU [2]. *Field-programmable gate arrays* (FPGAs) are arrays of reconfigurable logic and are popular devices for hardware prototyping. Recently, high-performance systems have begun to increasingly utilize FPGAs because of improvements in FPGA speeds and densities. The increasing cost of custom IC implementations along with improvements in FPGA tool flows has helped make FPGAs viable platforms for an increasing number of applications. *Graphics processing units* (GPUs) are designed to operate in a single instruction multiple data (SIMD) fashion. GPUs are being actively explored for general-purpose computations in recent times [4, 6, 5, 7]. The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data-intensive algorithms has arguably had a contribution in encouraging GPU manufacturers to design easily programmable general-purpose GPUs (GPGPUs). GPU architectures have been continuously evolving toward higher performance, larger memory sizes, larger memory bandwidths, and relatively lower costs.

Note that the hardware platforms discussed in this research monograph require an (expensive) communication link with the host processor. All the EDA applications considered have to work around this communication cost, in order to obtain a healthy speedup on their target platform. Future-generation hardware architectures may not face a high communication cost. This would be the case if the host and the accelerator are implemented on the same die or share the same physical RAM. However, for existing architectures, it is important to consider the cost of this communication while discussing the feasibility of the platform for a particular application.

1.2 EDA Algorithms Studied in This Research Monograph

In this monograph, we study two different categories of EDA algorithms, namely *control-dominated* and *control plus data parallel* algorithms. Our work demonstrates the rearchitecting of EDA algorithms from both these categories, to max-

imally harness their performance on the alternative platforms under consideration. We chose applications for which there is a strong motivation to accelerate, since they are used in key time-consuming steps in the VLSI design flow. Further, these applications have different degrees of inherent parallelism in them, which make them an interesting implementation challenge for these alternative platforms. In particular, Boolean satisfiability, Monte Carlo based statistical static timing analysis, circuit simulation, fault simulation, and fault table generation are explored.

1.2.1 Control-Dominated Applications

In the control-dominated algorithms category, this monograph studies the implementation of Boolean satisfiability (SAT) on the custom IC, FPGA, and GPU platforms.

1.2.2 Control Plus Data Parallel Applications

Among EDA problems with varying amounts of control and data parallelism, we accelerated the following applications using GPUs:

- Statistical static timing analysis (SSTA) using graphics processors
- Accelerating fault simulation on a graphics processor
- Fault table generation using a graphics processor
- Fast circuit simulation using graphics processor

1.3 Automated Approach for GPU-Based Software Acceleration

The key idea here is to partition a software subroutine into kernels in an automated fashion, such that multiple instances of these kernels, when executed in parallel on the GPU, can maximally benefit from the GPU's hardware resources. The software subroutine must satisfy the constraints that it (i) is executed many times and (ii) there are no control or data dependencies among the different invocations of this routine.

1.4 Chapter Summary

In recent times, improvements in VLSI system performance have slowed due to several *walls* that are being faced. Key among these are the power and memory walls. Since the growth of single-processor performance is hampered due to these walls, EDA software needs to explore alternate platforms, in order to deliver the increased performance required to design the complex electronics of the future.

In this monograph, we explore the acceleration of several different EDA algorithms (with varying degrees of inherent parallelism) on alternative hardware platforms. We explore custom ICs, FPGAs, and graphics processors as the candidate platforms. We study the architectural and performance tradeoffs involved in implementing several EDA algorithms on these platforms. We study two classes of EDA algorithms in this monograph: (i) control-dominated algorithms such as Boolean satisfiability (SAT) and (ii) control plus data parallel algorithms such as Monte Carlo based statistical static timing analysis, circuit simulation, fault simulation, and fault table generation. Another contribution of this monograph is to automatically generate GPU code to accelerate software routines that are run repeatedly on independent data.

This monograph is organized into four parts. In Part I of the monograph, different hardware platforms are compared, and the programming model used for interfacing with the GPU platform is presented. In Part II, we present techniques to accelerate a control-dominated algorithm (Boolean satisfiability). We present an IC-based approach, an FPGA-based approach, and a GPU-based scheme to accelerate SAT. In Part III, we present our approaches to accelerate control and data parallel applications. In particular we focus on accelerating Monte Carlo based SSTA, fault simulation, fault table generation, and model card evaluation of SPICE, on a graphics processor. Finally, in Part IV, we present an automated approach for GPU-based software acceleration. The monograph is concluded in Chapter 12, along with a brief description of next-generation hardware platforms. The larger goal of this work is to provide techniques to enable the acceleration of EDA algorithms on different hardware platforms.

References

1. A Platform 2015 Workload Model. http://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf
2. Denser, Faster Chips Deliver Knockout DSP Performance. http://electronicdesign.com/Articles/ArticleID̄0676
3. GPU Architecture Overview SC2007. http://www.gpgpu.org
4. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 47 (2004)
5. Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., Buck, I.: GPGPU: General-purpose computation on graphics hardware. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 208 (2006)
6. Owens, J.: GPU architecture overview. In: SIGGRAPH '07: ACM SIGGRAPH 2007 Courses, p. 2 (2007)
7. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Philips, J.C.: GPU Computing. In: Proceedings of the IEEE, vol. 96, pp. 879–899 (2008)

Chapter 2
Hardware Platforms

2.1 Chapter Overview

As discussed in Chapter 1, single-threaded software applications no longer obtain significant gains in performance with the current processor scaling trends. With the growing complexity of VLSI designs, this is a significant problem for the electronic design automation (EDA) community. In addition to multi-core processors, hardware-based accelerators such as custom-designed ICs, reconfigurable hardware such as FPGAs, and streaming processors such as graphics processing units (GPUs) are being investigated as a potential solution to this problem. These platforms allow the CPU to offload compute-intensive portions of an application to the hardware for a faster computation, and the results are transferred back to the CPU upon completion. Different platforms are best suited for different application scenarios and algorithms. The pros and cons of the platforms under consideration are discussed in this chapter.

The rest of this chapter is organized as follows. Section 2.2 discusses the hardware platforms studied in this monograph, with a brief introduction of custom ICs, FPGAs, and GPUs in Section 2.3. Sections 2.4 and 2.5 compare the hardware architecture and programming environment of these platforms. Scalability of these platforms is discussed in Section 2.6, while design turn-around time on these platforms is compared in Section 2.7. These platforms are contrasted for performance and cost of hardware in Sections 2.8 and 2.9, respectively. The implementation of floating point operations on these platforms is compared in Section 2.10, while security concerns are discussed in Section 2.11. Suitable applications for these platforms are discussed in Section 2.12. The chapter is summarized in Section 2.13.

2.2 Introduction

Most hardware accelerators are not stand-alone platforms, but are co-processors to a CPU. In other words, a CPU is needed for initial processing, before the compute-intensive task is off-loaded to the hardware accelerators. In some cases the hardware

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_2,
© Springer Science+Business Media, LLC 2010

accelerator might communicate with the CPU even during the computation. The different platforms for hardware acceleration in this monograph are compared in the following sections.

2.3 Hardware Platforms Studied in This Research Monograph

2.3.1 Custom ICs

Traditionally, custom ICs are included in a product to improve its performance. With a high production volume, the high manufacturing cost of the IC is easily amortized. Among existing hardware platforms, custom ICs are easily the fastest accelerators. By being application specific, they can deliver very high performance for the target application. There exist a vast literature of advanced circuit design techniques which help in reducing the power consumption of such ICs while maintaining high performance [36]. Some of the more well-known techniques to reduce power consumption (both dynamic and leakage) are design and protocol changes [31, 20], reducing supply voltage [17], variable Vt devices, dynamic bulk modulation [39, 40], power gating [18], and input vector control [25, 16, 41]. Also, newer gate materials which help achieve further performance gains at a low power cost are being investigated [32]. Due to their high performance and small footprint, custom ICs are the most suitable accelerators for space, military, and medical applications that are compute intensive.

2.3.2 FPGAs

A field-programmable gate array (FPGA) is an integrated circuit which is designed to be configured by the designer in the field. The FPGA is generally programmed using a hardware description language (HDL). The ability of the user to program the functionality of the FPGA in the field, along with the low non-recurring engineering costs (relative to a custom IC design), makes the FPGA an attractive platform for many applications. FPGAs have significant performance advantages over microprocessors due to their highly parallel architectures and significant flexibility. Hardware-level parallelism allows FPGA-based applications to operate 1 to 2 orders of magnitude faster than equivalent applications running on an embedded processor or even a high-end workstation. Compared to custom ICs, FPGAs have a somewhat lower performance, but their reconfigurability makes them an easy choice for several (particularly low-volume) applications.

2.3.3 Graphics Processors

General-purpose graphics processors turn the massive computational power of a modern graphics accelerator into general-purpose computing power. In certain

applications which include vector processing, this can yield several orders of magnitude higher performance than a conventional CPU. In recent times, general-purpose computation on graphics processors has been actively explored for several scientific computations [23, 34, 29, 35, 24]. The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data-intensive algorithms has arguably had a contribution in encouraging GPU manufacturers to design GPUs that are easy to program for general-purpose applications as well. GPU architectures have been continuously evolving toward higher performance, larger memory sizes, larger memory bandwidths, and relatively lower costs. Additionally, the development of open-source programming tools and languages for interfacing with the GPU platforms, along with the continuous evolution of the computational power of GPUs, has further fueled the growth of general-purpose GPU (GPGPU) applications.

A comparison of hardware platforms considered in this monograph is presented next, in Sections 2.4 through 2.12.

2.4 General Overview and Architecture

Custom-designed ICs have no fixed architecture. Depending on the algorithm, technology, target application, and skill of the designers, custom ICs can have extremely diverse architectures. This flexibility allows the designer to trade off design parameters such as throughput, latency, power, and clock speed. The smaller features also open the door to higher levels of system integration, making the architecture even more diverse.

FPGAs are high-density arrays of reconfigurable logic, as shown in Fig. 2.1 [14]. They allow a designer the ability to trade off hardware resources versus performance, by giving the hardware designers the choice to select the appropriate level of parallelism to implement an algorithm. The ability to tradeoff parallelism and pipelining yields significant architectural variety. The circuit diagram for a typical FPGA logic block is shown in Fig. 2.2, and it can implement both combinational and sequential logic, based on the value of the MUX select signal X. The lookup table (LUT) in this FPGA logic block is shown in Fig. 2.3. It consists of a 16:1 MUX circuit, implemented using NMOS passgates. This is the typical circuit used for implementing LUTs [30, 21]. The circuit for the 16 SRAM configuration bits (labeled as 'S' in Fig. 2.3) is shown in Fig. 2.4. The DFF of Fig. 2.2 is implemented using identical master and slave latches, each of which has an NMOS passgate connected to the clock and a pair of inverters in a feedback configuration to implement the storage element.

In the FPGA paradigm, the hardware consists of a regular array of logic blocks. Wiring between these blocks is achieved by reconfigurable interconnect, which can be programmed via passgates and SRAM configuration bits to drive these passgates (and thereby customize the wiring).

Recent FPGAs provide on-board hardware IP blocks for DSP, hard processor macros, and large amounts of on-chip block RAM (BRAM). These hardware IP

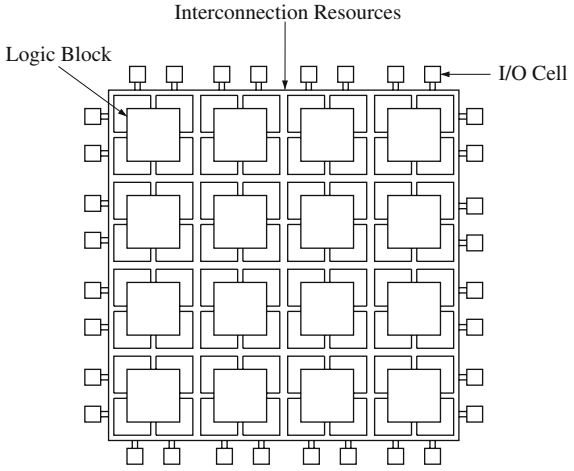


Fig. 2.1 FPGA layout [14]

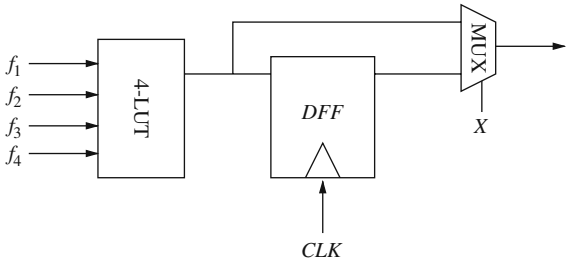


Fig. 2.2 Logic block in the FPGA

blocks allow a designer to perform many common computations without using FPGA logic blocks or LUTs, resulting in a more efficient design.

One downside of FPGA devices is that they have to be reconfigured every time the system is powered up. This requires the use of either a special external memory device (which has an associated cost and consumes real estate on the board) or an on-board microprocessor (or some variation of these techniques).

GPUs are commodity parallel devices which provide extremely high memory bandwidths and a large number of programmable cores. They can support thousand of simultaneously issued software threads operating in a SIMD fashion. GPUs have several multiprocessors which execute these software threads. Each multiprocessor has a special function unit, which handles infrequent, expensive operations, like divide and square root. There is a high bandwidth, low latency local memory attached to each multiprocessor. The threads executing on that multiprocessor can communicate among themselves using this local memory. In the current generation of NVIDIA GPUs, the local memory is quite small (16 KB). There is also a large global device memory (over 4 GB in some models) of GPU cards. Virtual memory is not implemented, and so paging is not supported. Due to this limitation,

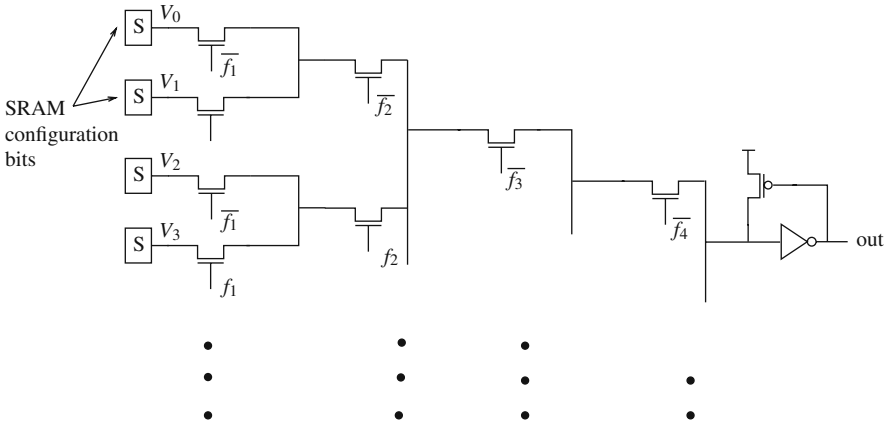


Fig. 2.3 LUT implementation using a 16:1 MUX

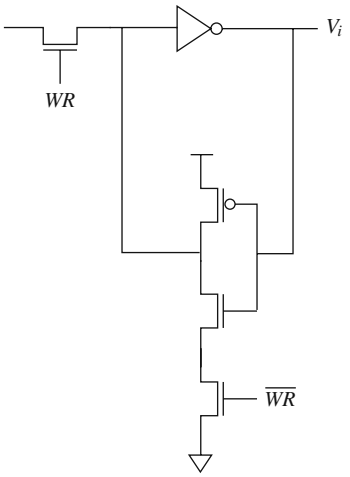


Fig. 2.4 SRAM configuration bit design

all the data has to fit in the global memory. The global device memory has very high bandwidth (but also has high latency) to the multiprocessors. The global device memory is not directly accessible by the host CPU nor is the host memory directly accessible to the GPU. Data from the host that needs to be processed by the GPU must be transferred via DMA (across an IO bus) from the host to the device memory. Similarly, data is transferred via DMA from the GPU to the CPU memory as well. GPU memory bandwidths have grown from 42 GB/s for the ATI Radeon X1800XT to 141.7 GB/s for the NVIDIA GeForce GTX 280 GPU [37]. A recent comparison of the performance in Gflops of GPUs to CPUs is shown in Fig. 2.5. A key drawback of the current GPU architectures (as compared to FPGAs) is that the on-chip memory cannot be used to store the intermediate data [22] of a

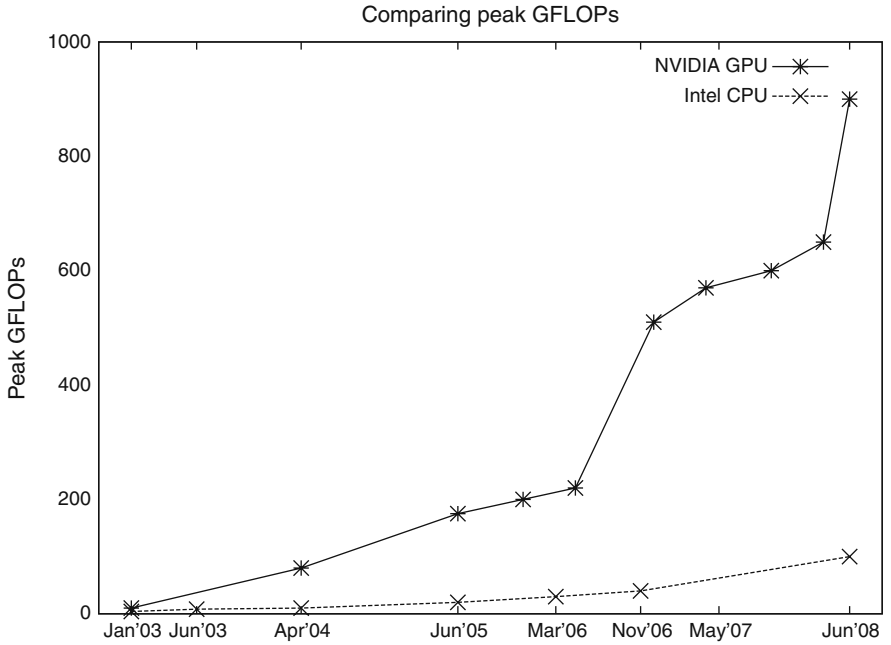


Fig. 2.5 Comparing Gflops of GPUs and CPUs [11]

computation. Only off-chip global memory (DRAM) can be used for storing intermediate data. On the FPGA, processed data can be stored in on-chip block RAM (BRAM).

2.5 Programming Model and Environment

Custom-designed ICs require several EDA tools in their design process. From functional correctness at the RTL/HDL level to the hardware testing and debugging of the final silicon, EDA tools and simulators are required at every step. For certain steps, a designer has to manually fix the design or interface signals to meet timing or power requirements. Needless to say, for ICs with several million transistors, design and testing can take months before the hardware masks are finalized for fabrication. Unless the design and manufacturing cost can be justified by large volumes or extremely high performance requirements, the custom design approach is typically not practical.

FPGAs are generally customized based on the use of SRAM configuration cells. The main advantage of this technique is that new design ideas can be implemented and tested much faster compared to a custom IC. Further, evolving standards and protocols can be accommodated relatively easily, since design changes are much simpler to incorporate. On the FPGA, when the system is first powered up, it

can initially be programmed to perform one function such as a self-test and/or board/system test, and it can then be reprogrammed to perform its main task. FPGA vendors provide software and hardware IP cores [3] that implement several common processing functions. More recently, high-end FPGAs have become available that contain one or more embedded microprocessors. Tasks that used to be performed by an external microprocessor can now be moved into the FPGA core. This provides several advantages such as cost reduction, significantly reduced data transfer times from FPGA to the microprocessor, simplified circuit board design, and a smaller, more power-efficient system. Debugging the FPGA is usually performed using embedded logic analyzers at the bitstream level [26]. FPGA debugging, depending on the design density and complexity, can easily take weeks. However, this is still a small fraction of the time taken for similar activities in the custom IC approach. Given these advantages, FPGAs are often used in low- and medium-volume applications.

In the recent high-level languages released for interfacing with GPUs, the hardware details of the graphics processor are abstracted away. High-level APIs have made GPU programming very flexible. Existing libraries such as ACML-GPU [2] for AMD GPUs and CUFFT and CUBLAS [4] for NVIDIA GPUs have inbuilt efficient parallel implementations of commonly used mathematical functions. CUDA [10] from NVIDIA provides guidelines for memory access and the usage of hardware resources for maximal speedup. Brook+ [2] from AMD-ATI provides a lower level API for the programmer to extract higher performance from the hardware. Further, GPU debugging and profiling tools are available for verification and optimization. In comparison to FPGAs or custom ICs, using GPUs as accelerators incurs a significantly lower design turn-around time.

General-purpose CPU programming has all the advantages of GPGPU programming and is a mature field. Several programming environments, debugging and profiling tools, and operating systems have been around for decades now. The vast amount of existing code libraries for CPU-based applications is an added advantage of system implementation on a general-purpose CPU.

2.6 Scalability

In high-performance computing, scalability is an important issue. Combining multiple ICs together for more computing power and using an array of FPGAs for emulation purposes are known techniques to enhance scalability. However, the extra hardware usually requires careful reimplementation of some critical portions of the design. Further, parallel connectivity standards (PCI, PCI-X, EMIF) often fall short when scalability and extensibility are taken into consideration.

Scalability is hard to achieve in general and should be considered during the architectural and design phases of FPGA-based or custom IC-based algorithm acceleration efforts. Scalability concerns are very specific to the algorithm being targeted, as well as the acceleration approach employed.

For graphics processors, existing techniques for scaling are intracluster and intercluster scaling. GPU providers such as NVIDIA and AMD provide multi-GPU solutions such as [12] and [1], respectively. These multi-GPU architectures claim high scalability, in spite of limited parallel connectivity, provided the application lends itself well to the architecture. Scalability requires efficient use of hardware as well as communication resources in multi-core architectures, custom ICs, FPGAs, and GPUs. Architecting applications for scalability remains a challenging open problem for all platforms.

2.7 Design Turn-Around Time

Custom ICs have a high design turn-around time. Even for modest sized designs, it takes many months from the start of the design to when the silicon is delivered. If design revisions are required, the cost and design turn-around time of custom ICs can become even higher.

FPGAs offer better flexibility and rapid prototyping capabilities as compared to custom designs. An idea or concept can be tested and verified in an FPGA without going through the long and expensive fabrication process of custom design. Further, incremental changes or design revisions (on an FPGA) can be implemented within hours or days instead of months. Commercial off-the-shelf prototyping hardware is readily available, making it easier to rapidly prototype a design. The growing availability of high-level software tools for FPGA design, along with valuable IP cores (prebuilt functions) for several commonly used control and signal processing tasks, makes it possible to achieve rapid design turn-arounds.

GPUs and CPUs allow for a far more flexible development environment and faster turn-around times. Newer compilers and debuggers help trace software bugs rapidly. Incremental changes or design revisions can be compiled much faster than in custom IC or FPGA designs. Code profiling technique for optimization purposes is a mature area [15, 10]. Thus, a software implementation can easily be used to rapidly prototype a new design or to modify an existing design.

2.8 Performance

Depending on the application, custom-designed ICs offer speedups of several orders of magnitude as compared to the single-threaded software performance on the CPU. However, as mentioned earlier, the time taken to design an IC can be prohibitive. FPGAs provide a performance that is intermediate between that of custom ICs and single-threaded CPUs. Hardware-level parallelism allows some FPGA-based applications to operate 1–2 orders of magnitude faster than an equivalent application running on a higher-end workstation. More recently, high-performance system designers have begun to explore the capabilities of FPGAs [28]. Advances in FPGA tool flows and the increasing FPGA speed and density characteristics

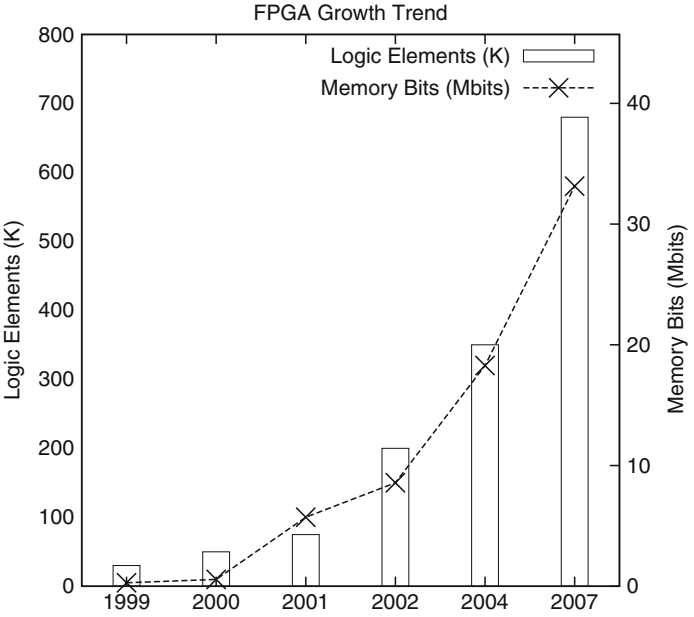


Fig. 2.6 FPGA growth trend [9]

(shown in Fig. 2.6) have made FPGAs increasingly popular. Compared to custom-designed ICs, FPGA-based designs yield lower performance, but the reconfigurable property gives it an edge over custom designs, especially since custom ICs incur significant NRE costs.

When measured in terms of power efficiency, the advantages of an FPGA-based computing strategy become even more apparent. Calculated as a function of millions of operations (MOPs) per watt, FPGAs have demonstrated greater than 1,000× power/performance advantages over today's most powerful processors [5]. For this reason, FPGA accelerators are now being deployed for a wide variety of power-hungry computing applications.

The power of the GPGPU paradigm stems from the fact that GPUs, with their large memories, large memory bandwidths, and high degrees of parallelism, are readily available as off-the-shelf devices, at very inexpensive prices. The theoretical performance of the GPU [37] has grown from 50 Gflops for the NV40 GPU in 2004 to more than 900 Gflops for GTX 280 GPU in 2008. This high computing power mainly arises due to a heavily pipelined and highly parallel architecture, with extremely high memory bandwidths. GPU memory bandwidths have grown from 42 GB/s for the ATI Radeon X1800XT to 141.7 GB/s for the NVIDIA GeForce GTX 280 GPU. In contrast, the theoretical performance of a 3 GHz Pentium4 CPU is 12 Gflops, with a memory bandwidth of 8–10 GB/s to main memory. The GPU IC is arguably one of the few VLSI platforms which has faithfully kept up with Moore's law in recent times. Recent CPU cores have 2–4 GHz core clocks, with single- and

multi-threaded performance capabilities. The Intel QuickPath Interconnect (4.8 GT/s version) copy bandwidth (using triple-channel 1,066 MHz DDR3) is 12.0 GB/s [7]. A 3.0 GHz Core 2 Quad system using dual-channel 1,066 MHz DDR3 achieves 6.9 GB/s. The level 2 and 3 caches have 10–40 cycle latencies. CPU cores today also support a limited amount of SIMD parallelism, with SEE [8] instructions.

Another key difference between GPUs and more general-purpose multi-core processors is hardware support for parallelism. GPUs have a hardware thread control unit that manages the distribution and assignment of thread blocks to multiprocessors. There is additional hardware support for synchronization within a thread block. Multi-core processors, on the other hand, depend on software and the OS to perform these tasks. However, the amount of power consumed by GPUs for executing only the accelerated portion of the computation is typically more than twice that needed by the CPU with all its peripherals. It can be argued that, since the execution is sped up, the power delay product (PDP) of a GPU-based implementation would potentially be lower. However, such a comparison is application dependent, and thus cannot be generalized.

2.9 Cost of Hardware

The non-recurring engineering (NRE) expense associated with custom IC design far exceeds that of FPGA-based hardware solutions. The large investment in custom IC development is easy to justify if the anticipated shipping volumes are large. However, many designers need custom hardware functionality for systems with low-to-medium shipping volumes. The very nature of programmable silicon eliminates the cost for fabrication and long lead times for chip assembly. Further, if system requirements change over time, the cost of making incremental changes to FPGA designs are negligible when compared to the large expense of redesigning custom ICs. The reconfigurability feature of FPGAs can add to the cost saving, based on the application. GPUs are the least expensive hardware platform for the performance they can deliver. Also, the cost of the software tool-chain required for programming GPUs is negligible compared to the EDA tool costs incurred by custom design and FPGAs.

2.10 Floating Point Operations

In comparison to software-based implementations, a higher numerical precision is a bigger problem for FPGAs and custom ICs. In FPGAs, for instance, on-chip programmable logic resources are utilized to implement floating point functionality for higher precisions [19]. These implementations consume significant die-area and tend to require deep pipelining before acceptable performance can be obtained. For example, hardware implementations of double precision multipliers typically require around 20 pipeline stages, and the square root operation requires 30–40 stages [38].

GPUs targeting scientific computations can handle IEEE double precision floating point [6, 13] while providing peak performance as high as 900 Gflops. GPUs, unlike FPGAs and custom ICs, provide native support for floating point operations.

2.11 Security and Real-Time Applications

In industry practice, design details (including HDL code) are typically documented to make reuse more convenient. At the same time, this makes IP piracy and infringement easier. It is estimated that the annual revenue loss due to IP infringement in the IC industry is in excess of $5 billion [42]. The goals of IP protection include enabling IP providers to protect their IPs against unauthorized use, protecting all types of design data used to produce and deliver IPs, and detecting and tracing the use of IPs [42].

FPGAs, because of their re-programmability, are becoming very popular for creating and exchanging VLSI IPs in the reuse-based design paradigm [27]. Existing watermarking and fingerprinting techniques embed identification information into FPGA designs to deter IP infringement. However, such methods incur timing and/or resource overheads and cause performance degradation. Custom ICs offer much better protection for intellectual property [33].

CPU/GPU software IPs have higher IP protection risks. The emerging trend is that most IP exchange and reuse will be in the form of soft IPs because of the design flexibility they provide. The IP provider may also prefer to release soft IPs and leave the customer-dependent optimization process to the users [27]. From a security point of view, protecting soft IPs is a much more challenging task than protecting hard IPs. Soft IPs are hard to trace and therefore not preferred in highly secure application scenarios.

Compared to a CPU/GPU-based implementation, FPGA and custom IC designs are truly *hard* implementations. Software-based systems like CPUs and GPUs, on the other hand, often involve several layers of abstraction to schedule tasks and share resources among multiple processors or software threads. The driver layer controls hardware resources and the operating system manages memory and processor utilization. For a given processor core, only one instruction can execute at a time, and hence processor-based systems continually run the risk of time-critical tasks pre-empting one another. FPGAs and custom ICs, which do not use operating systems, minimize these concerns with true parallel execution and dedicated hardware. As a consequence, FPGA and custom IC implementations are more suitable for applications that demand hard real-time computation guarantees.

2.12 Applications

Custom ICs are a good match for space, military, and medical compute-intensive applications, where the footprint and weight constraints are tight. Due to their high

performance, several DSP-based applications make use of custom-designed ICs. A custom IC designer can create highly efficient special functions such as arithmetic units, multi-port memories, and a variety of non-volatile storage units. Due to their cost and high performance, custom IC implementations are best suited for high-volume and high-performance applications.

Applications for FPGA are primarily hybrid software/hardware-embedded applications including DSP, video processing, robotics, radar processing, secure communications, and many others. These applications are often instances of implementing new and evolving standards, where the cost of designing custom ICs cannot be justified. Further, the performance obtained from high-end FPGAs is reasonable. In general, FPGA solutions are used for low-to-medium volume applications that do not demand extreme high performance.

GPUs are an upcoming field, but have already been used for accelerating scientific computations in fluid mechanics, image processing, and financial applications among other areas. The number of commercial products using GPUs is currently limited, but this might change due to newer architectures and high-level languages that make it easy to program the powerful hardware.

2.13 Chapter Summary

In recent times, due to the power, memory, and ILP walls, single-threaded applications do not see any significant gains in performance. Existing hardware-based accelerators such as custom-designed ICs, reconfigurable hardware such as FPGAs, and streaming processors such as GPUs are being heavily investigated as potential solutions. In this chapter we discussed these hardware platforms and pointed out several key differences among them.

In the next chapter we discuss the CUDA programming environment, used for interfacing with the GPUs. We describe the hardware, memory, and programming models for the GPU devices used in this monograph. This discussion is intended to serve as background material for the reader, to ease the explanation of the details of the GPU-based implementations of several EDA algorithms described in this monograph.

References

1. ATI CrossFire. http://ati.amd.com/technology/crossfire/features.html
2. ATI Stream Computing. http://ati.amd.com/technology/streamcomputing/sdkdwnld.html
3. CORE Generator System. http://www.xilinx.com/products/design-tools/logic-design/design-entry/coregenerator.htm
4. CUDA Zone. http://www.nvidia.com/object/cuda.html
5. FPGA-based hardware acceleration of C/C++ based applications. http://www.pldesignline.com/howto/201800344

6. Industry's First GPU with Double-Precision Floating Point. http://ati.amd.com/products/streamprocessor/specs.html
7. Intel Nehalem (microarchitecture). http://en.wikipedia.org/wiki/Nehalem-CPU-architecture
8. Intel SSE. http://www.tommesani.com/SSE.html
9. Mammoth FPGAs Require New Tools. http://www.gaterocket.com/device-native-verification/bid/7966/Mammoth-FPGAs-Require-New-Tools
10. NVIDIA CUDA Homepage. http://developer.nvidia.com/object/cuda.html
11. NVIDIA CUDA Introduction. http://www.beyond3d.com/content/articles/12/1
12. SLI Technology. http://www.slizone.com/page/slizone.html
13. Tesla S1070. http://www.nvidia.com/object/product-tesla-s1070-us.html
14. The Death of the Structured ASIC. http://www.chipdesignmag.com/print.php/articleId/434/issueId/16
15. Valgrind. http://valgrind.org/
16. Abdollahi, A., Fallah, F., Massoud, P.: An effective power mode transition technique in MTCMOS circuits. In: Proceedings, IEEE Design Automation Conference, pp. 13–17 (2005)
17. Bhavnagarwala, A.J., Austin, B.L., Bowman, K.A., Meindl, J.D.: A minimum total power methodology for projecting limits on CMOS GSI. IEEE Transactions Very Large Scale Integration Systems **8**(3), 235–251 (2000)
18. Bhunia, S., Banerjee, N., Chen, Q., Mahmoodi, H., Roy, K.: A novel synthesis approach for active leakage power reduction using dynamic supply gating. In: DAC '05: Proceedings of the 42nd Annual Conference on Design Automation, pp. 479–484 (2005)
19. Che, S., Li, J., Sheaffer, J., Skadron, K., Lach, J.: Accelerating compute-intensive applications with GPUs and FPGAs. In: Application Specific Processors, 2008. SASP 2008. Symposium on, pp. 101 – 107 (2008)
20. Chinnery, D.G., Keutzer, K.: Closing the power gap between ASIC and custom: An ASIC perspective. In: DAC '05: Proceedings of the 42nd Annual Design Automation Conference, pp. 275–280 (2005)
21. Chow, P., Seo, S., Rose, J., Chung, K., Paez-Monzon, G., Rahardja, I.: The design of a SRAM-based field-programmable gate array – part II : Circuit design and layout. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **7**(3), 321–330 (1999)
22. Cope, B., Cheung, P., Luk, W., Witt, S.: Have GPUs made FPGAs redundant in the field of video processing? In: Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on, pp. 111–118 (2005)
23. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 47 (2004)
24. Feng, Z., Li, P.: Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms. In: ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design, pp. 647–654. IEEE Press, Piscataway, NJ (2008)
25. Gao, F., Hayes, J.: Exact and heuristic approaches to input vector control for leakage power reduction. In: Proceedings, International Conference on Computer-Aided Design, pp. 527–532 (2004)
26. Graham, P., Nelson, B., Hutchings, B.: Instrumenting bitstreams for debugging FPGA circuits. In: FCCM '01: Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 41–50 (2001)
27. Jain, A.K., Yuan, L., Pari, P.R., Qu, G.: Zero overhead watermarking technique for FPGA designs. In: GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI, pp. 147–152 (2003)
28. Kuon, I., Rose, J.: Measuring the gap between FPGAs and ASICs. In: FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, pp. 21–30 (2006)

29. Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., Buck, I.: GPGPU: General-purpose computation on graphics hardware. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 208 (2006)
30. Mal, P., Cantin, J., Beyette, F.: The circuit designs of an SRAM based look-up table for high performance FPGA architecture. In: 45th Midwest Symposium on Circuits and Systems (MWCAS), vol. III, pp. 227–230 (2002)
31. Minana, G., Garnica, O., Hidalgo, J.I., Lanchares, J., Colmenar, J.M.: A power-aware technique for functional units in high-performance processors. In: DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design, pp. 456–459 (2006)
32. Molas, G., Bocquet, M., Buckley, J., Grampeix, H., Gély, M., Colonna, J.P., Martin, F., Brianceau, P., Vidal, V., Bongiorno, C., Lombardo, S., Pananakakis, G., Ghibaudo, G., De Salvo, B., Deleonibus, S.: Evaluation of HfAlO high-k materials for control dielectric applications in non-volatile memories. Microelectronic Engineering **85**(12), 2393–2399 (2008)
33. Oliveira, A.L.: Robust techniques for watermarking sequential circuit designs. In: DAC '99: Proceedings of the 36th ACM/IEEE Conference on Design Automation, pp. 837–842 (1999)
34. Owens, J.: GPU architecture overview. In: SIGGRAPH '07: ACM SIGGRAPH 2007 Courses, p. 2 (2007)
35. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Philips, J.C.: GPU Computing. In: Proceedings of the IEEE, vol. 96, pp. 879–899 (2008)
36. Raja, T., Agrawal, V.D., Bushnell, M.L.: CMOS circuit design for minimum dynamic power and highest speed. In: VLSID '04: Proceedings of the 17th International Conference on VLSI Design, p. 1035. IEEE Computer Society, Washington, DC (2004)
37. Schive, H.Y., Chien, C.H., Wong, S.K., Tsai, Y.C., Chiueh, T.: Graphic-card cluster for astrophysics (GraCCA) – performance tests. In: Submitted to NewAstronomy (2007)
38. Scrofano, R., G.Govindu, Prasanna, V.: A library of parameterizable floating point cores for FPGAs and their application to scientific computing. In: Proceedings of the 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms, pp. 137–148 (2005)
39. Wei, L., Chen, Z., Johnson, M., Roy, K., De, V.: Design and optimization of low voltage high performance dual threshold CMOS circuits. In: DAC '98: Proceedings of the 35th Annual Conference on Design Automation, pp. 489–494 (1998)
40. Yu, B., Bushnell, M.L.: A novel dynamic power cutoff technique DPCT for active leakage reduction in deep submicron CMOS circuits. In: ISLPED '06: Proceedings of the 2006 International Symposium on Low Power Electronics and Design, pp. 214–219 (2006)
41. Yuan, L., Qu, G.: Enhanced leakage reduction technique by gate replacement. In: DAC, pp. 47–50 (2005)
42. Yuan, L., Qu, G., Ghout, L., Bouridane, A.: VLSI design IP protection: solutions, new challenges, and opportunities. In: AHS '06: Proceedings of the First NASA/ESA Conference on Adaptive Hardware and Systems, pp. 469–476 (2006)

Chapter 3
GPU Architecture and the CUDA Programming Model

3.1 Chapter Overview

In this chapter we discuss the programming environment and model for programming the NVIDIA GeForce 280 GTX GPU, NVIDIA Quadro 5800 FX, and NVIDIA GeForce 8800 GTS devices, which are the GPUs used in our implementations. We discuss the hardware model, memory model, and the programming model for these devices, in order to provide background for the reader to understand the GPU platform better.

The rest of this chapter is organized as follows. We introduce the CUDA programming environment in Section 3.2. Sections 3.3 and 3.4 discuss the device hardware and memory models. The programming model is discussed in Section 3.5. Section 3.6 summarizes the chapter.

3.2 Introduction

Early computing systems were designed such that the rendering of the computer display was performed by the CPU itself. As displays became more complex, with higher resolutions and color depths, graphics accelerator ICs were developed to handle the graphics processing for computer displays. These ICs were initially quite primitive, with dedicated hardwired units to perform the display-rendering functionality. As more complex graphics abilities were demanded by the growing gaming industry, the first graphics processing units (GPUs) came into being, to replace the hardwired logic with a multitude of lightweight processors, each of which performed display manipulation of the computer display. These GPUs were natively designed as graphics accelerators for image manipulations, 3D rendering operations, etc. These graphics acceleration tasks require that the same operations are performed independently on different regions of the display. As a result, GPUs were designed to operate in a SIMD fashion, which is a natural computational paradigm for graphical display manipulation tasks.

Recently, GPUs are being actively exploited for general-purpose scientific computations [3, 5, 4, 6]. The growth of the general-purpose GPU (GPGPU) applications

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_3,
© Springer Science+Business Media, LLC 2010

stems from the fact that GPUs, with their large memories, large memory bandwidths, and high degrees of parallelism, are readily available as off-the-shelf devices, at very inexpensive prices. The theoretical performance of the GPU [7] has grown from 50 Gflops for the NV40 GPU in 2004 to more than 900 Gflops for GTX 280 GPU in 2008. This high computing power mainly arises due to a heavily pipelined and highly parallel architecture. The GPU IC is arguably one of the few VLSI platforms which has faithfully kept up with Moore's law in recent times. Further, the development of open-source programming tools and languages for interfacing with the GPU platforms has further fueled the growth of GPGPU applications.

CUDA (Compute Unified Device Architecture) is an example of a new hardware and software architecture for interfacing with (i.e., issuing and managing computations on) the GPU. CUDA abstracts away the hardware details and does not require applications to be mapped to traditional graphics APIs [2, 1]. CUDA was released by NVIDIA corporation in early 2007. The GPU device interacts with the host through CUDA as shown in Fig. 3.1.

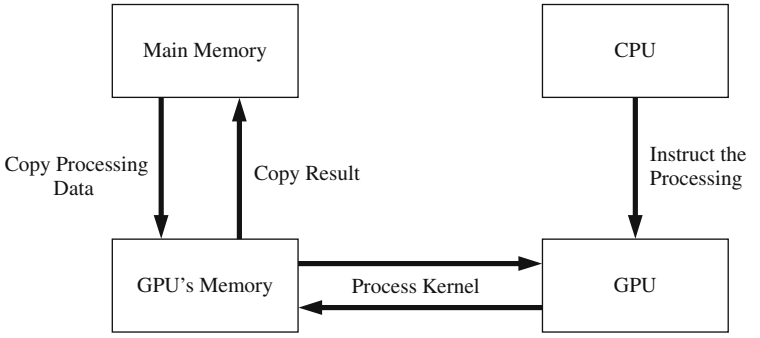


Fig. 3.1 CUDA for interfacing with GPU device

3.3 Hardware Model

As shown in Fig. 3.2, the GeForce 280 GTX architecture has 30 multiprocessors per chip and 8 processors (ALUs) per multiprocessor. The Quadro 5800 FX has the same hardware model as the 280 GTX device. The 8800 GTS, on the other hand, has 16 multiprocessors per chip. During any clock cycle, all the processors of a multiprocessor execute the same instruction, but may operate on different data. There is no mechanism to communicate *between* the different multiprocessors. In other words, no native synchronization primitives exist to enable communication between multiprocessors. We next describe the memory organization of the device.

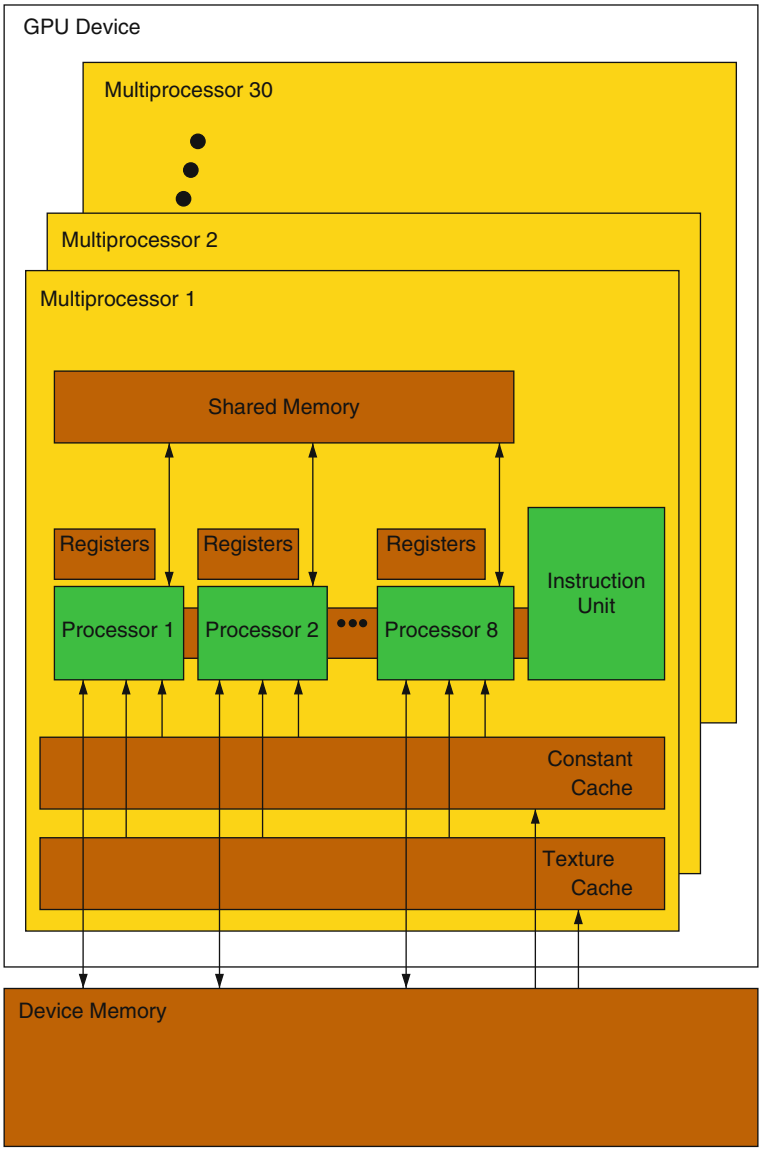


Fig. 3.2 Hardware model of the NVIDIA GeForce GTX 280

3.4 Memory Model

The memory model of NVIDIA GTX 280 is shown in Fig. 3.3. Each multiprocessor has on-chip memory of the following four types [2, 1]:

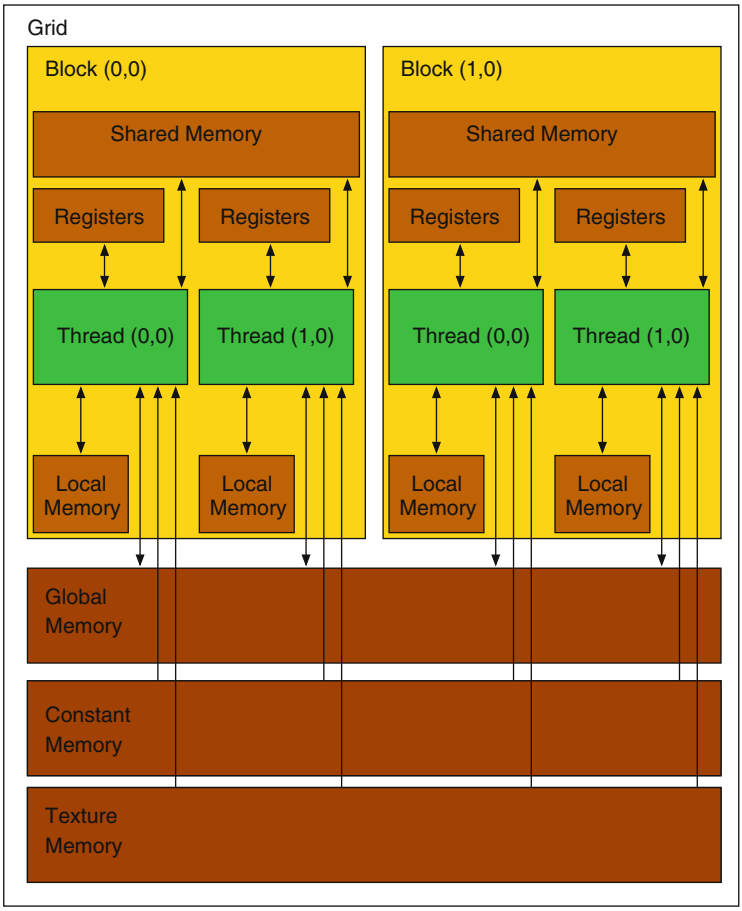


Fig. 3.3 Memory model of the NVIDIA GeForce GTX 280

- One set of local 32-bit *registers* per processor. The total number of registers per multiprocessor in the GTX 280 and the Quadro 5800 is 16,384, and for the 8800 GTS it is 8,192.
- A parallel data cache or *shared memory* that is shared by all the processors of a multiprocessor. The size of this shared memory per multiprocessor is 16 KB and it is organized into 16 banks.
- A read-only *constant cache* that is shared by all the processors in a multiprocessor, which speeds up reads from the constant memory space. It is implemented as a read-only region of device memory. The amount of constant memory available is 64 KB, with a cache working set of 8 KB per multiprocessor.
- A read-only *texture cache* that is shared by all the processors in a multiprocessor, which speeds up reads from the texture memory space. It is implemented as a read-only region of the device memory.

The local and global memory spaces are implemented as read–write regions of the device memory and are not cached. These memories are optimized for different uses. The local memory of a processor is used for storing data structures declared in the instructions executed on that processor.

The pool of shared memory within each multiprocessor is accessible to all its processors. Each block of shared memory represents 16 banks of single-ported SRAM. Each bank has 1 KB of storage and a bandwidth of 32 bits per clock cycle. Furthermore, since there are 30 multiprocessors on a GeForce 280 GTX or Quadro 5800 (GTS 8800), this results in a total storage of 480 KB (256 KB) per multiprocessor. For all practical purposes, this memory can be seen as a logical and highly flexible extension of the local memory. However, if two or more access requests are made to the same bank, a *bank conflict* results. In this case, the conflict is resolved by granting accesses in a serial fashion. Thus, shared memory must be accessed in a fashion such that bank conflicts are minimized.

Global memory is read/write memory that is *not* cached. A single floating point value read from (or written to) global memory can take 400–600 clock cycles. Much of this global memory latency can be hidden if there are sufficient arithmetic instructions that can be issued while waiting for the global memory access to complete. Since the global memory is not cached, access patterns can dramatically change the amount of time spent in waiting for global memory accesses. Thus, coalesced accesses of 32-bit, 64-bit, or 128-bit quantities should be performed in order to increase the throughput and to maximize the bus bandwidth utilization.

The texture cache is optimized for spatial locality. In other words, if instructions that are executed in parallel read texture addresses that are close together, then the texture cache can be optimally utilized. A texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. Device memory reads through *texture fetching* (provided in CUDA for accessing texture memory) present several benefits over reads from global or constant memory:

- Texture fetching is cached, potentially exhibiting higher bandwidth if there is locality in the (texture) fetches.
- Texture fetching is not subject to the constraints on memory access patterns that global or constant memory reads must respect in order to get good performance.
- The latency of addressing calculations (in texture fetching) is better hidden, possibly improving performance for applications that perform random accesses to the data.
- In texture fetching, packed data may be broadcast to separate variables in a single operation.

Constant memory fetches cost one memory read from device memory only on a cache miss, otherwise they just cost one read from the constant cache. The memory bandwidth is best utilized when *all* instructions that are executed in parallel access the same address of the constant memory. We next discuss the GPU programming and interfacing tool.

3.5 Programming Model

CUDA's programming model is summarized in Fig. 3.4. When programmed through CUDA, the GPU is viewed as a compute device capable of executing a large number of *threads* in parallel. Threads are the atomic units of parallel computation, and the code they execute is called a *kernel*. The GPU device operates as a coprocessor to the main CPU or host. Data-parallel, compute-intensive portions of applications running on the host can be off-loaded onto the GPU device. Such a portion is compiled into the instruction set of the GPU device and the resulting program, called a kernel, is downloaded to the GPU device.

A *thread block* (equivalently referred to as a *block*) is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronize their execution to coordinate memory accesses. Users can specify

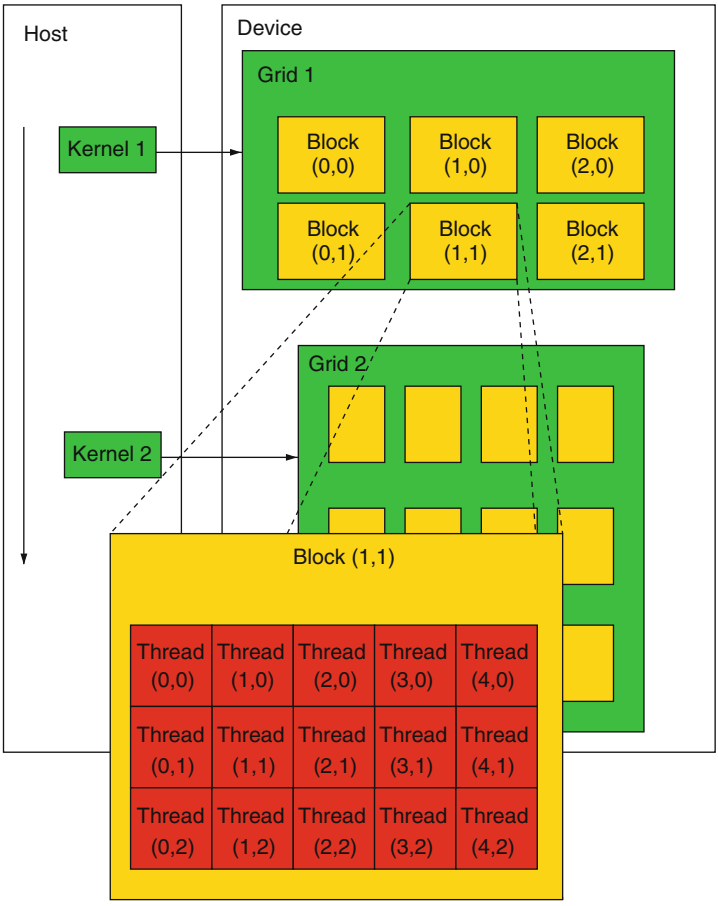


Fig. 3.4 Programming model of CUDA

synchronization points in the kernel, where threads in a block are suspended until they all reach the synchronization point. Threads are grouped in *warps*, which are further grouped in blocks. Threads have identifying numbers (*threadIDs*) which can be viewed as a one-, two-, or three-dimensional value. All the warps composing a block are guaranteed to run on the same multiprocessor and can thus take advantage of shared memory and local synchronization. Each warp contains the same number of threads, called the *warp size*, and is executed in a SIMD fashion; a *thread scheduler* periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources. In case of the NVIDIA GPUs discussed in this monograph, the warp size is 32. Thread blocks have restrictions on the *maximum* number of threads in them. The maximum number of threads grouped in a thread block, for all GPUs in this monograph, is 512. The number of threads in a thread block, *dimblock*, is decided by the programmer, who must ensure that (i) the maximum number of threads allowed in the block is 512 and (ii) the *dimblock* is a multiple of the warp size.

A thread block can be executed by a single multiprocessor. However, blocks of same dimensionality (i.e., orientation of the threads in them) and size (i.e., number of threads in them) that execute the same kernel can be batched together into a *grid* of blocks. The number of blocks in a grid is referred to as *dimgrid*. A grid of thread blocks is executed on the device by executing one or more blocks on each multiprocessor using time slicing. However, at a given time, at most 1,024 (768) threads can be active in a single multiprocessor on the 280 GTX or the Quadro 5800 (8800 GTS) GPU devices. When deciding the *dimblock* and *dimgrid* values, the restriction on the number of registers being used in a single multiprocessor has to be carefully monitored. If this limit is exceeded, the kernel will fail to launch.

In NVIDIA's current GPU devices, the synchronization paradigm is *local* to a thread block and is very efficient. However, threads belonging to different thread blocks of even the same grid cannot synchronize.

CUDA has several advantages over traditional GPGPU using graphics APIs. These are as follows:

- CUDA allows code to read from arbitrary addresses in memory – i.e., *scattered reads* are allowed.
- CUDA exposes a fast shared memory region (16 KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- CUDA allows faster downloads and readbacks to and from the GPU.
- CUDA supports integer and bitwise operations completely, including integer texture lookups.

The limitations of CUDA are as follows:

- CUDA uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.

- The double precision support has some deviations from the IEEE 754 standard. For example, only two IEEE rounding modes are supported (chop and round-to-nearest even). Also, the precision of division/square root is slightly lower than IEEE single precision.
- CUDA threads should be running in groups of at least 32 for best performance, with the total number of threads numbering in the thousands. If–else branches in the program code do not impact performance significantly, provided that each of the 32 threads takes the same execution path.
- CUDA-enabled GPUs are only available from NVIDIA (GeForce 8 series and above, Quadro, and Tesla).

3.6 Chapter Summary

In this chapter we discussed the hardware and memory models for the NVIDIA GPU devices used for experiments in this monograph. These devices are the GeForce 280 GTX, the Quadro 5800 FX, and the GeForce 8800 GTS. This discussion was provided to help the reader understand the details of our GPU-based algorithms described in the later chapters. We also described the CUDA programming model in detail, listing its advantages and disadvantages as well.

References

1. NVIDIA CUDA Homepage. `http://developer.nvidia.com/object/cuda.html`
2. NVIDIA CUDA Introduction. `http://www.beyond3d.com/content/articles/12/1`
3. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 47 (2004)
4. Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., Buck, I.: GPGPU: General-purpose computation on graphics hardware. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 208 (2006)
5. Owens, J.: GPU architecture overview. In: SIGGRAPH '07: ACM SIGGRAPH 2007 Courses, p. 2 (2007)
6. Pharr, M., Fernando, R.: GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley Professional, Boston, MA (2005)
7. Schive, H.Y., Chien, C.H., Wong, S.K., Tsai, Y.C., Chiueh, T.: Graphic-card cluster for astrophysics (GraCCA) – performance tests. In: Submitted to NewAstronomy (2007)

Part II
Control-Dominated Category

Outline of Part II

Part I of this monograph discussed the alternative hardware platforms being considered for accelerating EDA applications. In Part II of this monograph we focus on accelerating Boolean satisfiability (SAT) using these platforms. SAT is an example of an EDA application which is control dominated.

SAT is a classic NP-complete problem and has been widely studied in the past. Given a set V of variables, and a collection C of conjunctive normal form (CNF) clauses over V, the SAT problem consists of determining whether there is a satisfying truth assignment for C. Given the broad applicability of SAT to several diverse application domains such as logic synthesis, circuit testing, verification, pattern recognition, and others, there has been much effort devoted to devising efficient heuristics to solve SAT. In this monograph we present hardware solutions to the SAT problem, with the main goals of scalability and speedup.

Part II of this book is organized as follows. In Chapter 4, we discuss a custom IC-based hardware approach to accelerate SAT. In this approach, the traversal of the implication graph as well as conflict clause generation is performed in hardware, in parallel. We also propose a hardware approach to extract the minimum unsatisfiable core (i.e., the sub-formula consisting of the smallest set of clauses of the initial formula which is unsatisfiable) for any unsatisfiable formula. We store the clause literals in specially designed clause cells and implement the clauses in banks, such that clauses of variable widths can be accommodated in these banks. We also perform an up-front partitioning of the SAT problem in order to better utilize these banks. Our custom IC-based solution demonstrates significantly larger capacity than existing hardware SAT solvers and is scalable in the sense that several ICs can be effectively used to simultaneously operate on the same large SAT instance. We conducted layout and SPICE studies to estimate the area, power, and speed of this solution. Our approach has been functionally validated in Verilog. Our experiments show that instances with approximately 63K clauses can be accommodated on a single IC of size $1.5\,\text{cm} \times 1.5\,\text{cm}$. Our custom IC-based SAT solver results in over 3 orders of magnitude speed improvement over BCP-based software SAT approaches. Further, the capacity of our approach is significantly higher than all existing hardware-based approaches.

In Chapter 5, we discuss an FPGA-based hardware approach to accelerate SAT. In this approach, we store the clause literals in the FPGA slices. In order to solve large SAT instances, we partition the clauses into 'bins,' each of which can fit in the FPGA. This is done in a pre-processing step. In general, these bins may share variables and hence do not solve independent sub-problems. The FPGA operates on one bin at a time. All the bins of the partitioned SAT problem are stored in the on-chip block RAM (BRAM). The embedded PowerPC processor on the FPGA performs the task of loading the appropriate bin from the BRAM. Conflict clause generation and Boolean constant propagation (BCP) are performed in parallel in the FPGA hardware. The entire flow, which includes the preprocessing step, loading of the BRAM, programming the PowerPC, and the subsequent communication between partitions (which is required for BCP, conflict clause generation, and both inter- and intra-bin non-chronological backtracking), has been automated and verified for correctness on a Virtex-II Pro (XC2VP30) FPGA board. Experimental results and their analysis, along with the performance models, are discussed in detail. Our results demonstrate that an order of magnitude improvement in runtime can be obtained over the best-in-class software-based approach, by using a Virtex-4 (XC4VFX140) FPGA device. The resulting system can handle instances with as many as 10K variables and 280K clauses.

In Chapter 6, we present a SAT approach which employs a new GPU-enhanced variable ordering heuristic. In this approach, we augment a CPU-based complete procedure, with a GPU-based approximate procedure (which benefits from the high parallelism of the GPU). The CPU implements MiniSAT, while the GPU implements SurveySAT. The SAT instance is read and the search is initiated on the CPU. After a user-specified fraction of decisions have been made, the CPU invokes the GPU-based SurveySAT procedure multiple times and updates its variable ordering based on any decisions made by SurveySAT. This approach retains completeness (since it is implements a complete procedure) but has the potential of high speedup (since the approximate procedure is executed on a highly parallel graphics processor based platform). Experimental results demonstrate an average 64% speedup over MiniSAT, for several satisfiable and unsatisfiable benchmarks.

Chapter 4
Accelerating Boolean Satisfiability on a Custom IC

4.1 Chapter Overview

Boolean satisfiability (SAT) is a core NP-complete problem. Several heuristic software and hardware approaches have been proposed to solve this problem. In this work, we present a hardware solution to the SAT problem. We propose a custom IC to implement our approach, in which the traversal of the implication graph as well as conflict clause generation is performed in hardware, in parallel. Further, extracting the minimum unsatisfiable core (i.e., the formula consisting of the smallest set of clauses of the initial formula which is unsatisfiable) is also a computationally hard problem. Our proposed hardware approach, in addition to solving SAT, also efficiently extracts the minimum unsatisfiable core for any unsatisfiable formula. In our approach, clause literals are stored in specially designed clause cells. Clauses are implemented in banks, in a manner that allows clauses of variable width to be accommodated in these banks. To maximize the utilization of these banks, we initially partition the SAT problem. Our solution has significantly larger capacity than existing hardware SAT solvers and is scalable in the sense that several ICs can be used to simultaneously operate on the same SAT instance. Our area, power, and performance figures are derived from layout and SPICE (using extracted parasitics) estimates. The approach presented in this work has been functionally validated in Verilog. Experimental results demonstrate that our approach can accommodate instances with approximately 63K clauses on a single IC of size $1.5\,\text{cm} \times 1.5\,\text{cm}$. Our hardware-based SAT solving approach results in over 3 orders of magnitude speed improvement over BCP-based software SAT approaches (1–2 orders of magnitude over other hardware SAT approaches). The capacity of our approach is significantly higher than most hardware-based approaches. Further, the worst case power consumption was found to be ≤ 1 mW for our implementation.

The rest of this chapter is organized as follows. The motivation for this work is described in Section 4.2. Related previous approaches are discussed in Section 4.3. Section 4.4 describes the hardware architecture employed in our approach. It includes a discussion on the generation of implications and conflicts (which is done in parallel), along with the hardware partitioning utilized, the communication protocol that banks implement, and the generation of conflict-induced clauses.

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_4,
© Springer Science+Business Media, LLC 2010

An example of conflict clause generation is described in Section 4.5. Section 4.6 describes the up-front clause partitioning methodology, which targets maximum utilization of the hardware. Section 4.7 describes our approach to finding the unsatisfiable core. The experimental results we have obtained are reported in Section 4.8. Section 4.9 summarizes the chapter with some directions for future work in this area.

4.2 Introduction

Boolean satisfiability (SAT) [8] is a classic NP-complete problem, which has been widely studied in the past. Given a set V of variables, and a collection C of conjunctive normal form (CNF) clauses over V, the SAT problem consists of determining whether there is a satisfying truth assignment for C and reporting it. If no such assignment exists, C is called an unsatisfiable instance. A subset of C, such that this subset is also an unsatisfiable instance, is called an unsatisfiable core. *Formally, given a formula ψ, the formula ψ_C is an unsatisfiable core for ψ iff ψ_C is unsatisfiable and $\psi_C \subseteq \psi$*. Computing or extracting the minimum unsatisfiable core of a given unsatisfiable instance is also reported to be a computationally hard problem [26, 19].

Given the broad applicability of the SAT and the unsatisfiable core extraction problems to several diverse application domains such as logic synthesis, circuit testing, verification, pattern recognition, and others [13], there has been much effort devoted to devising efficient heuristics to solve them. Some of the more well-known software approaches for SAT include [28, 21, 11] and [16].

There has been much interest in the hardware implementation of SAT solvers as well. An excellent survey of existing hardware approaches to solve the SAT problem is found in [29]. Although several hardware implementations of SAT solvers have been proposed, there is, to the best of our knowledge, no hardware approach for extracting the unsatisfiable core. We therefore claim this work to be the first to present a hardware-based solution for minimum unsatisfiable core extraction.

Numerous applications can benefit from the ability to speedily obtain a small unsatisfiable core from an unsatisfiable Boolean formula. Applications like planning an assignment [18] can be cast as a SAT instance (equivalently referred to as a CNF instance in the sequel). The satisfiability of this instance implies that there exists a viable scheduling solution. On the other hand, if a planning is proven infeasible due to the SAT instance being unsatisfiable, a small unsatisfiable core can help in locating the reason for infeasibility. Similarly, an unsatisfiable instance in FPGA routing [23] implies that the channel is unroutable. A smaller unsatisfiable core in this case would be a geometrically smaller region, with potentially fewer routes, such that the routing is infeasible in this region. Quickly identifying the reason for unroutability is of importance in routing. Further, SAT-based unbounded model checking [20] also requires the efficient extraction of small unsatisfiable cores. Most approaches for extracting the unsatisfiable core are broadly based on the conflict analysis procedure described in [28].

The key motivation for using a hardware approach for SAT or unsatisfiable core extraction is speed. Therefore, in the bigger picture, the context in which our work would find its usefulness is one in which SAT checking or unsatisfiable core extraction is to be sped up, compared to the best-in-class software or hardware approaches. Our hardware-based SAT solver and unsatisfiable core extractor would be well suited for applications wherein the same instance or a slight modification of the instance is solved repeatedly. This property is found in applications like routing, planning, or SAT-based unbounded model checking, logic synthesis, VLSI testing, and verification. The cost of initial CNF partitioning and of loading the CNF instance onto the hardware is incurred only once, and the speedup obtained with repeated SAT solving would amply recover this cost. Even a modest speedup of such SAT-based algorithms is of great interest to the VLSI design automation community, since the fraction of the time spent performing SAT checks in these algorithms is very high.

A key requirement for any hardware approach for Boolean satisfiability or unsatisfiable core extraction is *capacity* and *scalability*. By the *capacity* of a hardware SAT approach, we mean the largest size of a SAT instance (in terms of number of clauses) that can fit in the hardware. Our proposed solution has significantly larger capacity than existing hardware-based solutions. In our approach, a single IC of size 1.5 cm $\times$ 1.5 cm can accommodate CNF instances containing $\sim$63,000 clauses (along with the logic required for solving the instance). This is significantly larger than the capacity of previous hardware approaches for Boolean satisfiability. By the *scalability* of a hardware SAT approach, we mean that multiple hardware SAT units can be easily made to operate in tandem, to tackle larger SAT instances.

In this work, we propose an approach that utilizes a custom IC to accelerate the SAT solution and the unsatisfiable core extraction processes, with the *goal of speedily solving large instances in a scalable fashion*. The hardware implements a variant of GRASP [28] (i.e., a slightly modified strategy of conflict-driven learning and non-chronological backtracking compared to [28]) . For the extraction of the unsatisfiable core, the hardware approach is augmented to implement the approach described in [19]. In this IC, literals and their complement are implemented as custom cells. Clauses of variable width are implemented in banks. Any row of a bank can potentially accommodate more than one clause. The SAT problem is mapped to this architecture in an initial partitioning step, in a manner that maximizes hardware utilization. Experimental results are obtained using area, power, and performance figures derived from layout and SPICE (using extracted layout-level parasitics) estimates. Our hardware approach performs, in parallel, both the tasks of implicit traversal of the implication graph and conflict clause generation. The contribution of this work is to come up with a high capacity, fast, scalable hardware SAT approach. We do not claim to propose any new SAT solution or unsatisfiable core extraction heuristics in this work. Note that although we used a variant of the BCP engine of GRASP [28] in our hardware SAT solver, the hardware approach can be modified to implement other BCP engines as well. The BCP logic of any BCP-based SAT solver can be ported to HDL and directly synthesized in our approach.

4.3 Previous Work

There have been several hardware-based SAT solvers reported in the literature, which are summarized and compared in [29]. Among these approaches, [34, 35] utilize configurable processors to accelerate SAT, demonstrating a maximum speedup of 60× using a board with 121 configurable processors. The largest example mapped to this structure had 24,700 clauses. In [37] and [38], the authors describe an FPGA-based SAT accelerator. The speedup obtained was 30×, with 64 FPGA boards required to handle an example containing 1,280 clauses. The largest example that the approach of [25] handles has about 1,300 clauses, with an average speedup of 10×. This work states that the hardware approaches reported in [4], [31], and [5] do not handle large SAT problems.

In [30] and [27], the authors present a software plus configurable hardware (configware) based approach to accelerate SAT. Software is used to do conflict diagnosis, backtrack, and clause management. Configware is used to do implication computation and next decision variable assignment. The speedup over GRASP [28] is between 1 and 2 orders of magnitude for the accelerated fraction of the SAT problem. The largest problem tackled has 214,304 clauses [27] (after conversion to 3-SAT, which can double the number of clauses [30]). In contrast, our approach performs *all* tasks in hardware, with a corresponding speedup of 1–2 orders of magnitude over the existing hardware approaches, as shown in the sequel. In most of the above approaches, the capacity of the proposed approaches is clearly limited, and scalability is a significant problem. The approach in this work is inspired by the requirement of handling significantly larger problems on a single die and also with the need to allow the design to scale more elegantly. By utilizing a custom IC approach, a single die can accommodate significantly larger SAT instances than most of what the above approaches report.

The previous approaches for the extraction of an unsatisfiable core have been software-based techniques. The complexity of this problem has been well studied and algorithms have been reported in [7, 9, 10] and [26]. Some of the proposed solutions with experimental data to support their algorithms include [6], in which an adaptive search is conducted, guided by clauses' hardness. References [12, 24] and [33] report resolution-based techniques for generating the empty clause. The unsatisfiable core reported in these cases is the set of clauses involved in the derivation of the empty clause. The minimum unsatisfiability prover from [15] improves upon the existing approaches by removing unnecessary clauses from unsatisfiable sub-formulas to make them minimal.

The approach in [19] attempts to find the *minimum* unsatisfiable core for a given formula. The augmentation of our hardware architecture for extracting the unsatisfiable core is in accordance with this approach. Broadly speaking, [19] employs a SAT solver to search for the minimum unsatisfiable core. This allows a natural match to our hardware-based SAT engine. Resolution-based techniques for unsatisfiable core extraction are not a natural fit to our approach, since resolution is inherently a serial process.

Extended abstracts of the work described in this chapter can be found in [32, 14].

4.4 Hardware Architecture

We next discuss the hardware architecture of our approach, starting with an overview.

4.4.1 Abstract Overview

Figure 4.1 shows an abstract view of our approach, to illustrate the main concept, and to explain how Boolean constraint propagation (BCP) [28] is carried out. Note that the physical implementation we use is different from this abstracted view, as subsequent sections will describe. In Fig. 4.1, the clause bank stores all clauses (a maximum of n clauses on m variables). In the hardware there are $n \cdot m$ *clause cells*, each of which stores a single literal of the SAT instance. The bank architecture is capable of implicitly storing the implication graph and consequently generating implications and conflicts. A variable is assigned by the *decision engine* and the assignment is communicated to the clause bank via the *base cells*. The clause bank, in turn, generates implications and possible conflicts due to this assignment. *This is done in parallel, at hardware speeds*. The base cells sense these implications and conflicts and in turn communicate them back to the decision engine. The decision engine accordingly assigns the next variable or, in case of a conflict, generates a conflict-induced clause and backtracks non-chronologically [28].

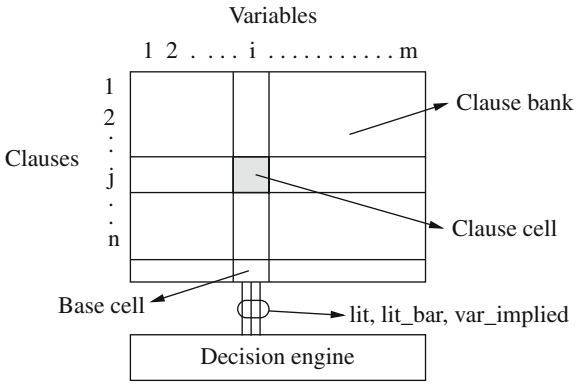


Fig. 4.1 Abstracted view of the proposed idea

As seen in Fig. 4.1, a column in the bank corresponds to a variable, a row corresponds to a clause, and a *clause cell* corresponds to a literal (which can be positive, negative, or absent) in the clause. The clause cell is central to our idea and provides the parallelism obtainable by solving the satisfiability problem in hardware.

The overall flow for solving any SAT instance S consists of first loading S into the clause bank. The hardware then solves S, after which a new SAT instance may be loaded and solved.

4.4.2 Hardware Overview

The actual hardware architecture of our SAT IC differs from the abstracted view of the previous section. The differences are not functional, rather they are induced by circuit partitioning and speed considerations. The different components of the hardware SAT IC are briefly described next.

The hardware details are presented in the following order. The finite state machine for the decision engine is explained in Section 4.4.3.1. The core circuit structure of our implementation, the clause cell, is capable of computing the implication graph implicitly and also helps in generating implications and conflicts, all in parallel. This is explained in Section 4.4.3.2. The implications and conflicts are sensed and forwarded to the decision engine by the base cells. The base cell and its interaction with the decision engine are explained in Section 4.4.3.3. In practice, we do not have a single clause bank as shown in Fig. 4.1. Rather, clauses are arranged in several banks, with a limited number of rows (clauses) and columns (variables). Each bank has several *strips*, which partition the columns of the bank into smaller groups. Between strips, we have special cells which allow us to implement arbitrarily long rows (clauses). The bank and strip structures are explained in Section 4.4.3.4. Because we partition the hardware into many banks, it is possible that a particular variable occurs in several banks. Therefore, implications or assignments on such variables, generated in a bank b_i, must be communicated to other banks b_j where the same variable occurs. This communication is performed by a hierarchical arrangement of communication units, arranged in a tree fashion. The details of this inter-bank communication are provided in Section 4.4.3.5. Figure 4.2 describes the banks and the inter-bank communication units. It also shows the centrally located BCP engine, as well as the banks for storing conflict-induced clauses.

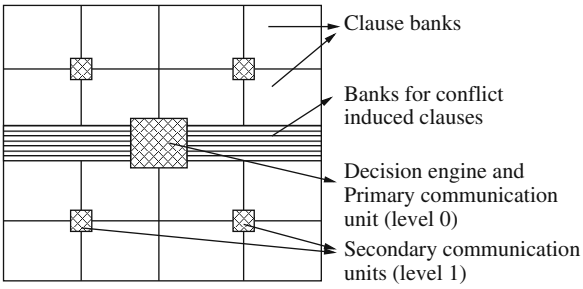

Clause banks
Banks for conflict induced clauses
Decision engine and Primary communication unit (level 0)
Secondary communication units (level 1)

Fig. 4.2 Generic floorplan

4.4.3 Hardware Details

4.4.3.1 Decision Engine

Figure 4.3 shows the state machine of the decision engine. To begin with, the CNF instance is loaded onto the hardware. Our hardware uses dynamic circuits so all signals are initialized into their precharged or predischarged states (in the *refresh* state). The decision engine assigns the variables in the order of their *identification tag*, which is a numerical ID for each variable, statically assigned such that most commonly occurring variables are assigned a lower tag. The decision engine assigns a variable (in *assign_next_variable* state) and this assignment is forwarded to the banks via the base cells. The decision engine then waits for the banks to compute all the implications during *wait_for_implications* state. If no conflict is generated due to the assignment, the decision engine assigns the next variable. If there is a conflict, all the variables participating in the conflict clause are communicated by the banks to the decision engine via the base cell. Based on this information, during the *analyze_conflict* state, the base cell generates the conflict-induced clause and then stores it in the clause bank. Also it non-chronologically backtracks according to the GRASP [28] algorithm. Each variable in a bank retains the decision level of the current assignment/implication. When the backtrack level is lower than this stored decision level, then the stored decision level is cleared before further action by the decision engine during the *execute_conflict* state. After a conflict is analyzed, the banks are again refreshed (in the *precharge* state) and the backtracked decision is applied to the banks. If all the variables have either been assigned or implied with no conflicts (this is detected from the assignment on the last level), the CNF instance is reported to be 'satisfiable' (in the *satisfied* state of the decision engine finite state

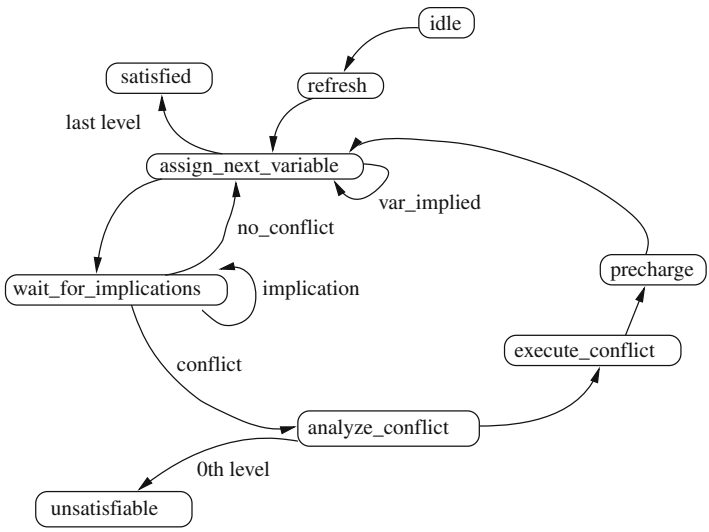


Fig. 4.3 State diagram of the decision engine

machine). On the other hand, if the decision engine has already backtracked on the variable at the 0th level and a conflict still exists, the CNF instance is reported to be 'unsatisfiable' (in the *unsatisfiable* state).

4.4.3.2 Clause Cell

Figure 4.4 shows the signal interface of a clause cell. Figure 4.5 provides details of the clause cell structure. Each column (variable) in the bank has three signals – *lit*, *lit_bar*, and *var_implied*, which are used to communicate assignments, implications, and conflicts on that variable. Each row (clause) in the bank has a signal *clausesat_bar* to indicate if the clause is satisfied. The 2-bit *free_lit_cnt* signals serve as an indicator of the number of free literals in the clause. If the literal in the clause cell is free (indicated by *iamfree*) then *out_free_lit_cnt* is one more than *in_free_lit_cnt*. The *imp_drv* and *cclause_drv* signals facilitate generation of implications and conflict clauses, respectively. Also, each row has a *termination cell* at its end (which we assume is at the right side of the row) which drives the *imp_drv* and *cclause_drv* signals. We next describe the encoding of these signals and how they are employed to perform BCP.

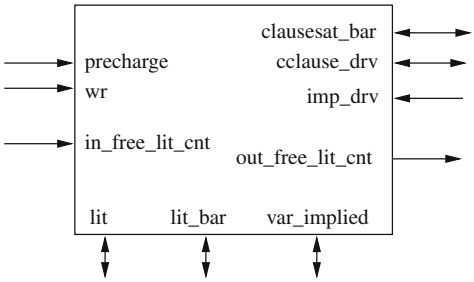


Fig. 4.4 Signal interface of the clause cell

Note that the signals *lit*, *lit_bar*, *var_implied*, and *cclause_drv* are predischarged and *clausesat_bar* is a precharged signal. Also, each clause cell has two single-bit registers namely *reg* and *reg_bar* to store the literal of the clause. The data in these registers can be driven in or driven out on the *lit* and *lit_bar* signals.

A variable is said to *participate* in a clause if it appears as a positive or negative literal in the clause. The encoding of the *reg* and *reg_bar* bits is as shown in Table 4.1. The *iamfree* signal for a variable indicates that the variable has not been assigned a value yet, nor has it been implied.

The assignments and failure-driven assertions [28] are driven on *lit*, *lit_bar*, and *var_implied* signals by the decision engine whereas implications are driven by the clause cells. Communication in both directions (i.e., from clause cell to the decision engine and vice versa) is performed via the base cells using the above signals. There exists a base cell for each variable. Table 4.2 lists the encoding of the *lit*, *lit_bar*, and *var_implied* signals.

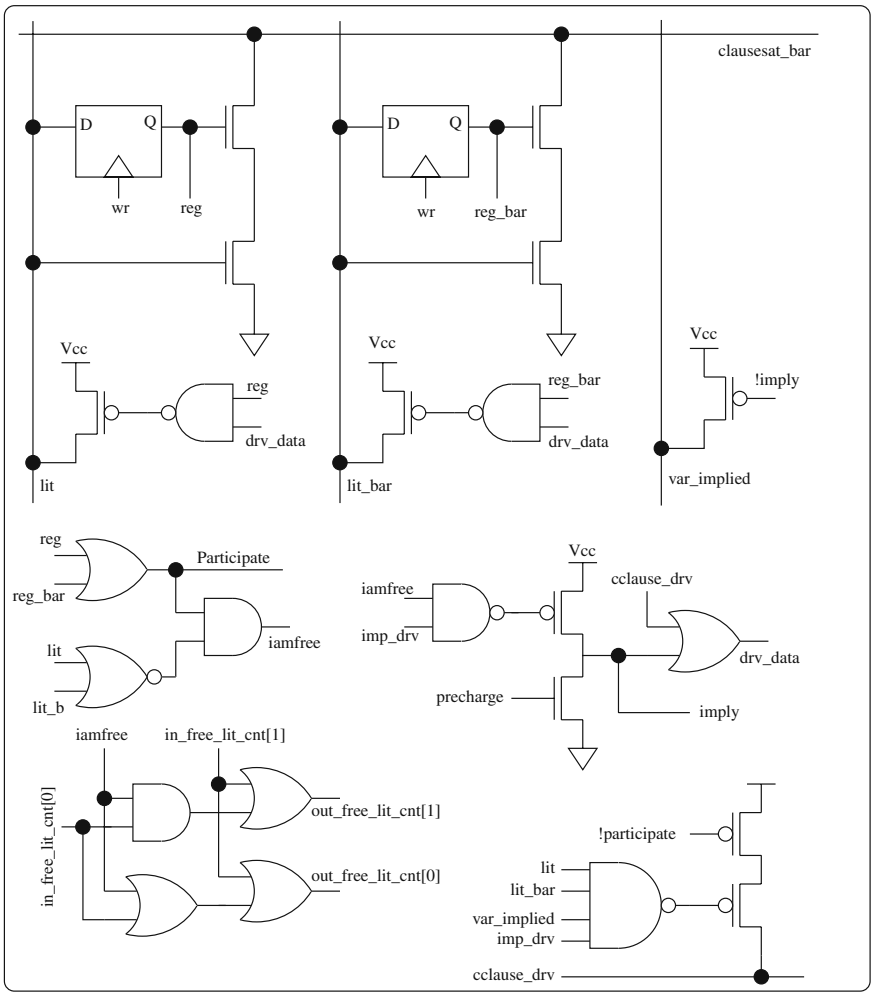


Fig. 4.5 Schematic of the clause cell

Table 4.1 Encoding of {*reg*,*reg_bar*} bits

Encoding	Meaning
00	Variable does not participate in clause
10	Variable participates as a positive literal
01	Variable participates as a negative literal
11	Illegal

If a variable V_i participates in clause C_j and no value has been assigned or implied on the *lit* and *lit_bar* signals for V_i, then V_i is said to contribute a *free literal* to

Table 4.2 Encoding of {*lit*,*lit_bar*} and *var_implied* signals

Encoding		Meaning
00	0	Variable is neither assigned nor implied
01	0	Value 0 is assigned to the variable
10	0	Value 1 is assigned to the variable
01	1	Value 0 is implied on the variable
10	1	Value 1 is implied on the variable
11	1	0 as well as 1 implied, i.e., conflict
11	0	Variable participates in conflict-induced clause
00	1	Illegal

clause C_j. This is indicated by the assertion of the signal *iamfree* for the (j,i)th clause cell. Also, a clause is satisfied when variable V_i participates in clause C_j and the value on the *lit* and *lit_bar* signals for V_i matches the register bits in clause cell c_{ji}. In such a case, the precharged signal *clausesat_bar* for C_j is pulled down by c_{ji}.

If clause C_j has only one free literal and C_j is unsatisfied, then C_j is called a *unit clause* [28]. When C_j becomes a unit clause with c_{ji} as the only free literal, its *termination cell* senses this condition by monitoring the value of *free_lit_cnt* and testing if its value is 1. If *free_lit_cnt* is found to be 1, the termination cell asserts the *imp_drv* signal. When c_{ji} (which is the free literal cell) senses the assertion of *imp_drv*, then it drives out its *reg* and *reg_bar* values on the *lit* and *lit_bar* wires and also asserts its *var_implied* signal, indicating an implication on variable V_i.

A conflict is indicated by the assertion of the *cclause_drv* signal. It can be asserted by the termination cell or a clause cell. The termination cell asserts *cclause_drv* when *free_lit_cnt* indicates that there is no free literal in the clause and the clause is unsatisfied (indicated by *clausesat_bar* staying precharged). A participating clause cell c_{ji} asserts *cclause_drv* for clause C_j when it detects a conflict on variable V_i and senses *imp_drv*. When *cclause_drv* is asserted for clause C_j, all the clause cells in C_j drive out their respective *reg* and *reg_bar* values on the respective *lit* and *lit_bar* wires. In other words the *drv_data* signal for the (j,i)th clause cell is asserted (or *reg* and *reg_bar* are driven out on *lit* and *lit_bar*) when either (i) *cclause_drv* is asserted or (ii) *imp_drv* is asserted, and the current clause cell has its *iamfree* signal asserted. Thus, if two clauses cause different implications on a variable, both clauses will drive out all their literals (which will both be high, since *lit* and *lit_bar* are predischarged signals). This indicates a conflict to the decision engine, which monitors the state of *lit*, *lit_bar*, and *var_implied* for each variable. This can trigger a chain of *cclause_drv* assertions leading to backtracking of the implication graph *in parallel*, which causes all the variables taking part in the conflict clause to be identified.

Figure 4.6 shows the layout view of our clause cell. The layout, generated in a full-custom manner, had a size of 12 μm by 9 μm and was implemented in a 0.1 μm technology.

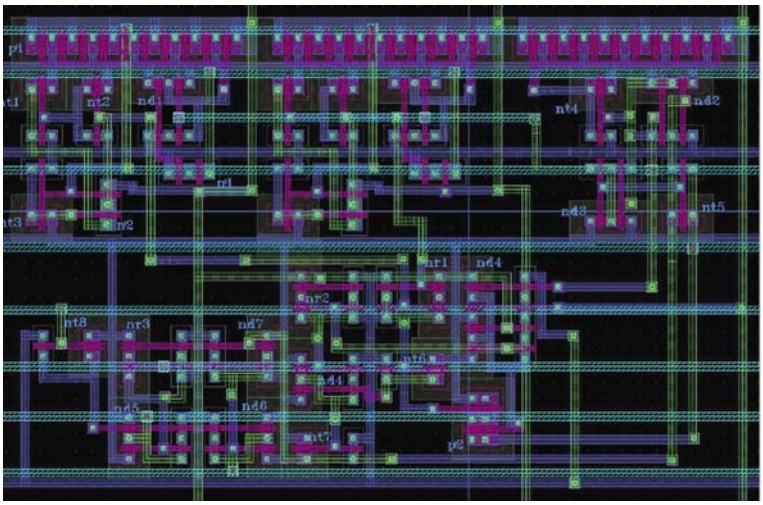


Fig. 4.6 Layout of the clause cell

4.4.3.3 Base Cell

There is one base cell for each variable in a bank. The base cell performs several functions. It stores information about its variable (its *identification tag*, value, decision level, and assigned/implied state). It also detects an implication on the variable, participates in generating the conflict-induced clause, and helps in performing non-chronological backtrack. These aspects of the base cell functionality are discussed next, after an explanation of its signal interface.

- Signal Interface
  Figure 4.7 shows the signal interface of the base cell. The signals *lit*, *lit_bar*, and *var_implied* in the base cell are bidirectional and are the means of communication between the decision engine and the clause bank. This communication is directed by the base cell. The signal *curr_lvl* stores the value of the current decision level. The base cell of each variable keeps track of any decision or implication on its

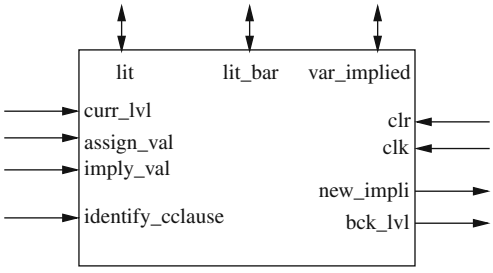


Fig. 4.7 Signal interface of the base cell

variable through the signals *assign_val* and *imply_val*, respectively. The signal *identify_cclause* is used during conflict analysis as described later. The *bck_lvl* signal indicates the level that the engine backtracks to, in case of a conflict. The *new_impli* signal is driven when an implication is detected.

- Detecting Implications
  Figure 4.8 shows the circuitry in the base cell to generate the *new_impli* signal, which is high for one clock cycle when an implication occurs (this constraint is required for the decision engine to remain in the state *wait_for_implications* while there are any new implications (indicated by *new_impli*)). This is done as follows. Initially both the flip-flop outputs are low. When the *var_implied* signal is high during the positive edge of a clock pulse, the flip-flop labeled *A* has its output driven high. This causes the output of the AND gate feeding the wired-OR to be driven high. In the next clock pulse, the flip-flop labeled *B* has its output driven high. This signal pulls the output of the AND gate (feeding the wired-OR) low. Thus, due to a *var_implied* signal, the *new_impli* is high for exactly one clock pulse. The flip-flops are cleared using the *clr* signal which is controlled by the decision engine. The *clr* is asserted during the *refresh* state for all base cells and during the *execute_conflict* state (for base cells having a decision level higher than the current backtrack level *bck_lvl*).

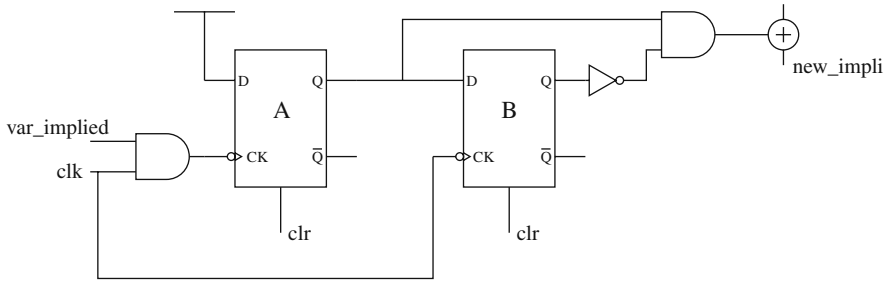


Fig. 4.8 Indicating a new implication

- Conflict Clause Generation
  The base cell also has the logic to identify a conflict clause literal and appropriately communicate it to the clause banks (for the purpose of creating a new conflict clause). During the *analyze_conflict* state, the decision engine sets the *identify_cclause* signal high. The base cell then records the current values of *lit*, *lit_bar*, and *var_implied*. If the tuple is equal to 110, the base cell drives the complement of this variable to the clause bank and asserts the clause write signal (*wr*) for the next available clause. This ensures that the conflict clause is written into the clause bank. Thus, any variable participating in the current conflict and having its *lit*, *lit_bar*, and *var_implied* as 110 is recorded and hence the conflict-induced clause is generated.
  As the conflict-induced clauses are generated dynamically, the width of the conflict clause banks cannot be fixed while programming the CNF instance in the

hardware. Therefore, the width of conflict-induced clause banks is kept equal to the number of variables in the given CNF instance. The decision engine can still pack more than one conflict-induced clause in one row of the conflict clause banks. To be able to use the space in the conflict-induced clause banks effectively, we propose to store only the clauses having fewer literals than a predetermined limit, updated in a first-in-first-out manner (such that old clauses are replaced by newly generated clauses). Further, we can utilize the clause banks for regular or conflict clauses, allowing our approach to devote a variable number of banks for conflict clauses, depending on the SAT instance.

- Non-chronological Backtrack
  The decision level to which the SAT solver backtracks, in case of a conflict, is determined by the base cell. The schematic for this logic is described next. Figure 4.9 shows the circuitry in the base cell to determine the backtrack level [28]. The signal *my_lvl* is the decision level associated with the variable. The signal *bck_lvl* (backtrack level) is a wired-OR signal. The variable which has the highest decision level among all the variables participating in a conflict sets the value of *bck_lvl* to its *my_lvl*. This is done as follows. Let the set of variables participating in the conflict be called C. Let $v^{\max}$ be the variable with the highest decision level among all variables $v \in C$. Each bit of every variable v's decision level is XNORed with the corresponding bit of the current value of *bck_lvl*. If the most significant bits *my_lvl*[k] and *bck_lvl*[k] are equal (which makes the output of the corresponding XNOR high) then the output of the XNOR of the next most significant bits is checked and so on. If for a certain bit i, *my_lvl*[i] is low and *bck_lvl*[i] is high, then the value of *bck_lvl* is higher than this variable's *my_lvl*. The output of the XNOR of the rest of the lesser significant bits ($j < i$) for this variable is ignored. This is done by ANDing the output of the ith bit's XNOR with the *my_lvl*[$i-1$] bit, to get a '0' result which is wire-ORed into *bck_lvl*[$i-1$]. This in turn gets trickled down to the *my_lvl* of the least significant bit. On the other hand, in case *my_lvl*[i] is high and *bck_lvl*[i] is low, then the AND gate feeding the wired-OR for the ith bit would drive a high value to the wired-OR and hence update *bck_lvl*[i] to high. The above continues until all the bits of *bck_lvl* are equal to the corresponding bits of $v^{\max}$'s decision level.

Our hardware SAT solver, consisting of clause banks, clause cells, base cells, decision engine, conflict generation, BCP, and non-chronological backtracking, has been implemented in Verilog and has been simulated and verified for correctness.

4.4.3.4 Partitioning the Hardware

In a typical CNF instance, a very small subset of variables participate in a single clause. Thus, putting all the clauses in one monolithic bank, as shown in the abstracted view of the hardware (Fig. 4.1), results in a lot of non-participating clause cells. For the DIMACS [1] examples, on average, more than 99% of the clause cells do not participate in the clauses if we arrange all the clauses in one bank. Therefore we partition the given CNF instance into disjoint subsets of clauses and put each

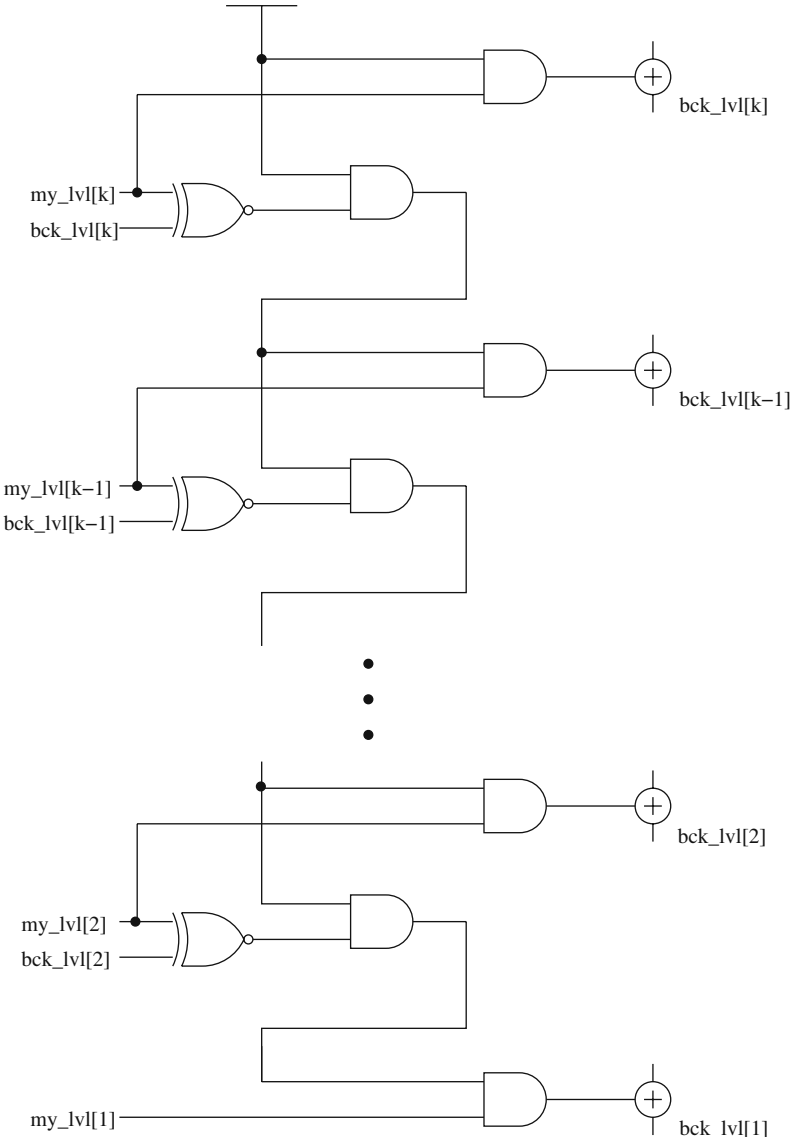


Fig. 4.9 Computing backtrack level

subset in a separate clause bank. Though a clause is fully contained in one bank, note that a variable may appear in more than one banks.

Figure 4.10 depicts an individual bank. Each bank is further divided into *strips* to facilitate a dense packing of clauses (such that the non-participating clause cells are minimized). We try to fit more than one clause per row with the help of strips. This is achieved by inserting a column of *terminal cells* between the strips. Figure 4.11

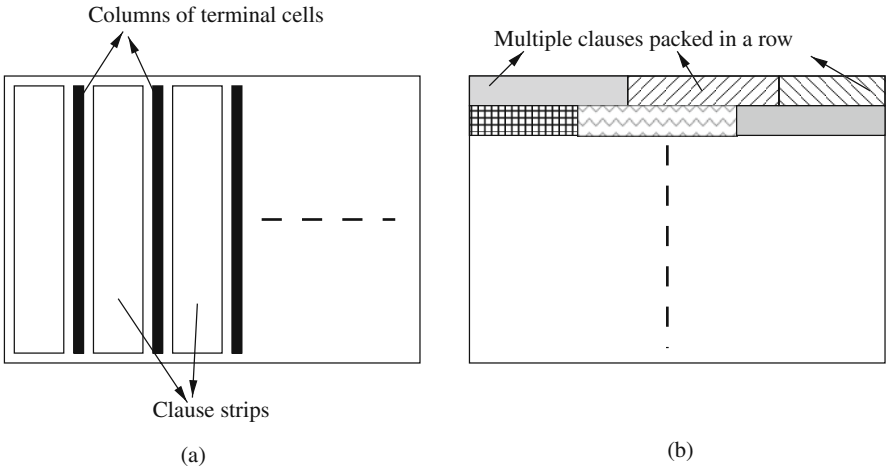


Fig. 4.10 **(a)** Internal structure of a bank. **(b)** Multiple clauses packed in one bank-row

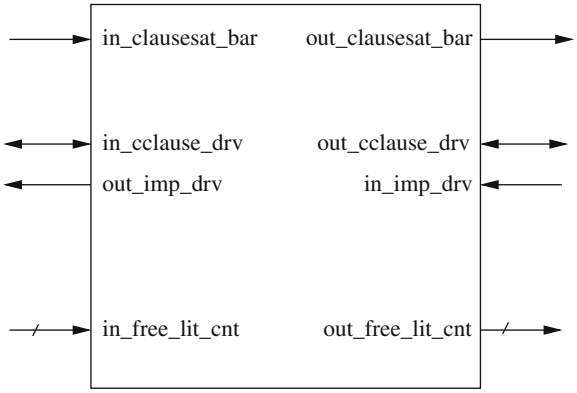


Fig. 4.11 Signal interface of the terminal cell

describes the signal interface of the terminal cell, while Fig. 4.12 shows the detailed schematic of the terminal cell. Each terminal cell has a programmable register bit indicating if the cell should act as a mere connection between the strips or act as a clause termination cell. While acting as a connection, the terminal cell repeats the *clausesat_bar*, *cclause_drv*, *imp_drv*, and *free_lit_cnt* signals across the strips, thereby expanding a clause over multiple strips. However, while acting as a clause termination cell, it generates *imp_drv* and *cclause_drv* signals for the clause being terminated. A new clause can start from the next strip (the strip to the right of the terminal cell).

The number of clause cell columns in a bank (or a strip) is called the width of a bank (or a strip) and the number of rows in a bank is called the height of a bank.

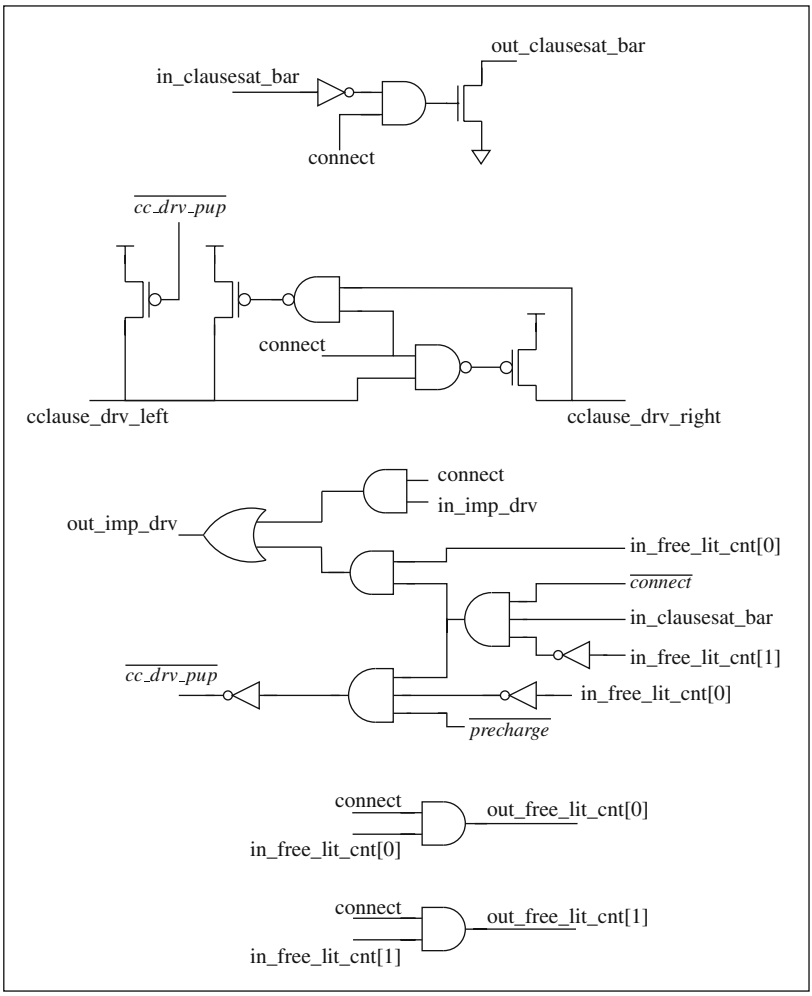


Fig. 4.12 Schematic of a terminal cell

On the basis of extensive experimentation, we settled on 25 rows and 6 columns in a strip. With the help of terminal cells, we can connect *as many strips as needed in a bank*. Consequently, a bank will have 25 rows *but its width is variable since the bank can have any number of strips connected to each other through the terminal cells*.

The algorithm for partitioning the problem into banks and for packing the clauses of any bank into its strips (to minimize the number of non-participating cells) is described in Section 4.6. Also, experimental results and optimal dimensions of the banks and strips are presented in Section 4.8.

4.4.3.5 Inter-bank Communication

Since a variable may appear in multiple banks (we refer to such variables as *repeated variables*), implications on such variables need to be communicated between the banks. Also, the assignments done by the decision engine need to be communicated to the banks and the implications or conflict clauses generated in the bank need to be communicated back to the decision engine.

In our design, we employ a hierarchical arrangement of *communication units* to perform this communication between the banks and the decision engine, as depicted in Fig. 4.13. Each column in the bank has a *base cell* that actually drives and senses the *lit*, *lit_bar*, and *var_implied* signals for that variable and communicates with the decision engine through a hierarchy of communication units. As seen in Fig. 4.13, the communication units and base cells form a tree structure. The communication unit directly interacting with the decision engine is said to be at 0th *level* of hierarchy and base cells are said to be at the highest level of hierarchy.

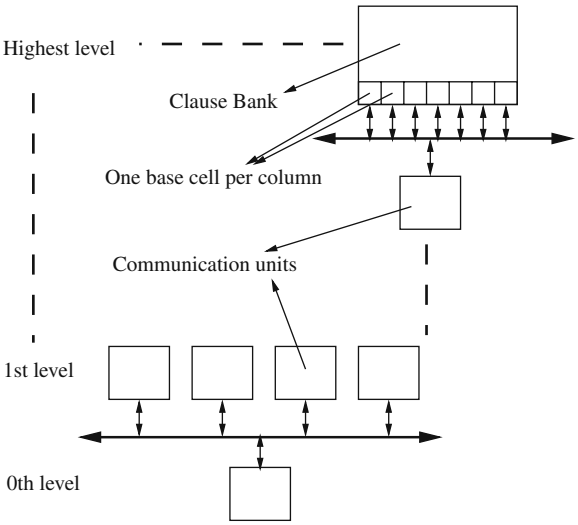


Fig. 4.13 Hierarchical structure for inter-bank communication

Each variable is associated with an *identification tag* as explained in Section 4.4.3.1. Every base cell has a register to store the identification tag of the variable it represents. The base cells and the decision engine use the identification tags to communicate assignments, implications, conflict clause variables, and backtrack level. A base cell also has a programmable register bit named *repeat bit* and a register named *repeat level*. The repeat bit indicates if the variable represented by the base cell is a repeated variable. The repeat level register for any variable v is pre-programmed with the hierarchy level of the communication unit that forms the root of the subtree containing all the base cells containing that repeated variable v. If the repeat bit for variable v is set, and an implication has occurred on v, the base cell

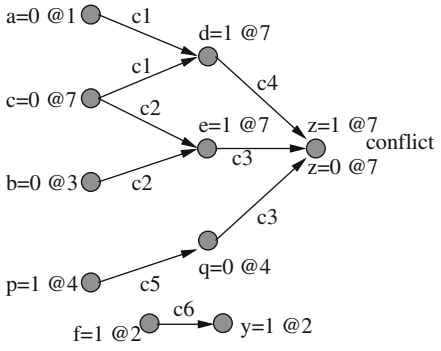
of the variable v communicates the implied value, its identification tag, and its repeat level to the communication unit C at the next lower level of hierarchy. The communication unit C communicates these data to other communication units at lower levels if the repeat level of the implied variable v is lower than its own hierarchy level. In this way, the inter-bank implication communication is completed using the smallest possible communication subtree, allowing for maximal parallelism during inter-bank communication.

The assignments made by the decision engine are broadcast to all levels. The variables participating in the conflict-induced clause are also communicated to the decision engine via this hierarchy.

Figure 4.2 shows the proposed floorplan. The decision engine is at the center of the chip surrounded by the clause banks. Additional banks required to store the conflict-induced clauses are also near the center of the chip. Each communication unit resides at the center of the chip area occupied by the banks in its communication subtree, as shown in Fig. 4.2.

4.5 An Example of Conflict Clause Generation

Figure 4.14 shows an example CNF instance, its implication graph, and how it is implicitly traversed in this scheme. $c_1, \ldots, c_6$ are the clauses as shown in Fig. 4.14b. Let us call the *lit*, *lit_bar*, and *var_implied* signals for a variable as a *signal triplet*. Initially all signal triplets are predischarged and held at high impedance. The implication graph in Fig. 4.14a shows a conflict occurring at decision level 7. $a = 0$, $b = 0$, $p = 1$, and $f = 1$ are the assignments made before level 7 and $q = 0$ and $y = 1$ are the implications caused by them. Figure 4.14c shows the transitions occurring on the signal triplet of each variable. Decisions are reflected as logic low and implication as logic high on the *var_implied* signal. The decision $c = 0$ at level 7 causes implications on d and e due to clauses c_1 and c_2, respectively. It results in c_3 and c_4 imposing conflicting requirements on the value of z. Therefore, c_3 drives 011 and c_4 drives 101 on the signal triplet of z, and the resultant status on z becomes 111. Note that triplet signals that are 0 are initially predischarged, so that they can be driven to 1 during the implication graph analysis. After the occurrence of a conflict, an implicit process of back-traversal of the graph starts in hardware. The conflict on z causes the assertion of the *cclause_drv* signal in c_3 and c_4 which in turn causes the data in their registers to be driven on the *lit* and *lit_bar* signals. Thus, 111 gets driven on the signal triplets of d due to c_4, and e and q due to c_3 (as they are implied variables). The 111 on d causes the assertion of *cclause_drv* in c_1, resulting in 110 on a and c as they are decision variables. Similarly 110 is driven on b and c due to c_2 and on p due to c_5. And thus the variables taking part in the conflict clause are a, b, c, and p and the conflict clause is formed by inverting their assigned values, i.e., $(a + b + c + \bar{p})$. Also, it can be seen that the status on f and y does not change as they are not part of the conflict graph. Thus implications and conflict clauses are implicitly generated and in parallel, and hence the process is quite fast.



(a) Implication Graph

c_1	$(a + c + d)$
c_2	$(b + c + e)$
c_3	$(\bar{z} + \bar{e} + q)$
c_4	$(\bar{d} + z)$
c_5	$(\bar{p} + \bar{q})$
c_6	$(\bar{f} + y)$

(b) CNF instance

	a	b	c	d	e	f	p	q	y	z
Initial (predischarge)	000	000	000	000	000	000	000	000	000	000
Assignments till @7	010	010				100	100			
Implications till @7								011	101	
Assignment @7			010							
Implications @7				101	101					
Conflict @7										111
Backtracking				111	111			111		111
Conflict clause variables	110	110	110				110			

(c) Implicit, Parallel Generation of Conflict Induced Clause

Fig. 4.14 Example of implicit traversal of implication graph

In case of multiple conflicts, our approach would create a single conflict clause which is the disjunction of all the new conflict clauses. This leads to lesser pruning of the search space as compared to storing the new conflict clauses individually.

In the current form, our hardware SAT solver only records the last row of the table (only the variables with decisions) in the conflict clause. A possible extension of our approach for generating smaller clauses (with fewer literals) is to store a row which is below the row corresponding to the conflict (i.e., row 7 of Figure 4.14c) and has the smallest number of entries (excluding the entry for the variable on which the conflict is detected). For example, the literals of row 8 of Figure 4.14c would yield a conflict clause $(\overline{d} + \overline{e} + q)$. Variable z would not be added in this conflict clause since it is the variable on which the conflict is detected. Adding this variable would not help in pruning the search space efficiently.

4.6 Partitioning the CNF Instance

This section describes the algorithms used to partition the given CNF instance into banks and strips. We cast these problems as hypergraph partitioning problems and use hMetis [17] to solve them.

To partition the CNF instance into multiple banks, we represent the clauses as vertices in the hypergraph and variables as hyperedges. Let $C = c_1,c_2,\ldots,c_n$ be the set of all clauses and $V = v_1,v_2,\ldots,v_m$ be the set of all variables in the given CNF instance. Then the resultant hypergraph is $G = (U,E)$, where $U = u_1,u_2,\ldots,u_n$ is a set of n vertices each corresponding to a clause in C and $E = e_1,e_2,\ldots,e_m$ is a set of m hyperedges each corresponding to a variable in V. Edge e_i connects vertex u_j if and only if variable v_i participates in clause c_j. This hypergraph is partitioned with hMetis such that each balanced partition contains k vertices and the number of hyperedges cut due to partitioning is minimized.

To partition a bank into strips, we represent the clauses as hyperedges and variables as vertices in the hypergraph. Similar to the above construction, let $C_i = c_1,c_2,\ldots,c_k$ be the set of clauses and $V_i = v_1,v_2,\ldots,v_l$ be the set of variables in bank B_i. Then the resultant hypergraph is $G_i = (U_i,E_i)$, where $U_i = u_1,u_2,\ldots,u_l$ is a set of l vertices each corresponding to a variable in V_i and $E_i = e_1,e_2,\ldots,e_k$ is a set of k hyperedges each corresponding to a clause in C_i. Edge $e_p \in E_i$ connects vertex $u_q \in U_i$ if and only if variable v_q participates in clause c_p.

After each bank is partitioned into strips, we need to order the strips so as to minimize the number of rows required to fit the clauses in the bank. For this purpose, we use a two-dimensional graph bandwidth minimization heuristic along with a greedy bin packing approach to pack the clauses in the rows. Figure 4.10b illustrates the packing of multiple clauses in one row. We perform bandwidth minimization on the matrix corresponding to the clauses of a bank. The bandwidth minimization problem consists of finding a permutation of the rows (clauses) and the columns (literals) of a matrix that keeps all the non-zero elements in a band that is as close as possible to the main diagonal. We use the following heuristic approach to perform bandwidth minimization.

For each clause C_i in the strip, we assign it a *gravity* $G(C_i)$ which is computed as follows: $G(C_i) = \sum_{C_j \in R(C_i)} (P(C_j) \cdot S(C_i,C_j))$.

Here, $R(C_i)$ is the set of clauses which have at least one variable common with clause C_i and $P(C_j)$ is the index of the current row of C_j and $S(C_i,C_j)$ is the number of common variables between clauses C_i and C_j.

The exact dual is used for computing the gravity of every variable in the current strip. The pseudocode for the bandwidth minimization algorithm is shown in Algorithm 1.

As shown in Algorithm 1, we alternate the gravity computation and rearrangement between clauses and variables. With every rearrangement of clauses and variables within bank s in an increasing order of gravity, we compute a new cost. The cost of the arrangement is the number of rows required to fit the clauses (of bank s). The greedy bin packing step simply packs the rearranged clauses of a bank into its rows, such that each clause uses an integral number of strips.

Algorithm 1 Pseudocode for Bandwidth Minimization

```
Best_Cost = Infinity
for i = 1; i ≤ Number of iterations; i++ do
    Compute gravity of all clauses in bank s
    Rearrange clauses in increasing order of gravity
    Compute gravity of all variables in bank s
    Rearrange variables in increasing order of gravity
    Perform greedy bin packing of clauses into strips
    Compute cost of current arrangement Cost_i
    if (Best_Cost ≥ Cost_i) then
        Best_Cost = Cost_i
        Store current arrangement
    end if
end for
return(Stored Arrangement)
```

4.7 Extraction of the Unsatisfiable Core

The work in [19] proposes a SAT-based algorithm for computing the minimum unsatisfiable core. The approach of [19] in brief is as follows: Given a Boolean formula ψ defined over n variables, $X = x_1,...,x_n$, such that ψ has m clauses, $\Omega = \omega_1,...,\omega_m$, the approach begins with the definition of a set S of m new variables $S = s_1,...,s_m$, and the creation of a new formula ψ' defined on $n+m$ variables, $X \cup S$, with m clauses $\Omega' = \omega_1',...,\omega_m'$. Each clause $\omega_i' \in \psi'$ is derived from a corresponding clause $\omega_i \in \psi$ as $\omega_i' = \neg s_i + \omega_i$. For a certain assignment to the variables in S, ψ' can be satisfiable or unsatisfiable. The *minimum* unsatisfiable core is obtained from the unsatisfiable sub-formula with the least number of S variables assigned to value 1.

The model of [19] can be seamlessly implemented in our hardware architecture. This is because this model simply extends the SAT problem. Since our approach exploits the parallelism which is inherent in any SAT problem, the two approaches can be naturally integrated. The experimental results reported in [19] are strongly limited by the number of variables and clauses in the problem instances. Although they compute the minimum unsatisfiable core, which was not reported by earlier approaches, the complexity of the model is significant for a software-based SAT solver. Testing on bigger instances was limited due to the inability of software SAT solvers to handle such instances. This is where our hardware-based SAT solver fits in. It elegantly complements their approach by providing a fast and scalable SAT solver to find the unsatisfiable core. Pseudocode for this algorithm is shown in Algorithm 2.

The following changes are made to our architecture to implement the above approach. In order to introduce the set S of m new variables (m is the initial number of clauses), the number of base cells is increased by m. The *identification tag* of any new variables (which is also the decision level of the new variables) is set to be lower than all the variables in the original SAT instance. Also since we add a

Algorithm 2 Pseudocode for Extracting the Minimum Unsatisfiable core

```
min_unsat_core(ψ(X,Ω)){
S ← add_new_variables(|Ω|) // add variables s1, s2, ... ,sm
ψ' ← Φ
for i = 1;i ≤ |Ω|;i++ do
    ω'i ← ¬si + ωi
    ψ' ← ψ' ∪ ω'i
end for
min_clause_solve(ψ') // explained in text
}
```

new variable to each clause, we have to add a new clause cell in each of the m clauses. Since we use efficient SAT instance partitioning, clause bank partitioning, and clause packing techniques, the overhead in terms of new clause cells required is $\leq m^2$. The extraction procedure (*min_clause_solve*($\psi^{'}$)) for the unsatisfiable core proceeds as follows. We perform repeated invocations of the hardware SAT solver with a different set of variables $S^{'} \subseteq S$ being assigned to 1. For a certain run, prior to the first assignment made by the decision engine, the signals *lit*, *lit_bar*, and *var_implied* for all the variables in $S^{'}$ are driven to 100 (i.e., forcing a decision of 1 on all variables $s_i \in S^{'}$). If the SAT solver reports the SAT instance as unsatisfiable, the clauses containing $s_i \in S^{'}$ are recorded. The corresponding clauses of the original SAT instance together make one unsatisfiable core. Next, a new clause consisting of all the variables in $S^{'}$ is added to the clause bank in a manner similar to adding a conflict-induced clause. In other words, we add a clause $\sum(\neg s_i)$, where $s_i \in S^{'}$, to the instance. This new clause avoids generating the same unsatisfiable core in future runs. Amongst all the unsatisfiable cores, the core with the smallest number of clauses is the minimum unsatisfiable core and is finally reported.

Other existing optimization techniques which are discussed in [19] can also be easily grafted in the modified hardware SAT solver. For example, any conflict-induced clause containing only variables $s_i \in S$ also generates an unsatisfiable core. This is because the clauses of the original SAT instance, corresponding to the clauses which contain s_i, represent an unsatisfiable core and can be recorded.

4.8 Experimental Results

To validate our ideas, we tested several examples from the DIMACS [1] test suite and from the SAT-2004 [3] competition benchmark suite. The examples we used are listed in Table 4.3, along with the number of clauses and variables (columns 1 through 3). For an IC of size 1.5 cm on a side, we can accommodate 1.875 million clause cells. The total number of strips in the IC is therefore 12,500. The IC implements a total of six hierarchical levels in the inter-bank communication methodology.

We tested the functionality of the clause and termination cells, the implication generation, and conflict clause generation logic in Verilog. The chip-level perfor-

Table 4.3 Partitioning and binning results

Instance	#Clauses	#Vars	PF (initial)	PF (opt.)	#Strips	Avg #Strips per cl.
par16-3	3,344	1,014	379	9.53	486	1.93
ii8b4	8,214	1,067	474	14.68	1,548	2.19
am	7,814	2,268	835	8.42	1,021	2.04
par32-5	10,325	3,175	1,183	9.01	1,426	1.76
ii16a1	19,368	1,649	719	25.71	10,514	2.87
ii32c4	20,862	758	137	12.45	8,178	4.57
dekker	58,308	19,472	8,346	10.40	8,084	1.78
frg2mul	62,943	10,313	3,063	8.68	10,514	2.41

mance estimates were obtained by running SPICE [22], using layout-extracted parasitics. The hardware SAT IC was implemented in a 0.1 μm process, with a VDD of 1.2 V.

For all the examples listed in Table 4.3, we performed partitioning (into banks) and binning (into strips) as described in Section 4.6. The initial partitioning was performed to create banks with 200 clauses. We define the *packing factor* (PF) as a figure of merit for the partitioning and binning procedure:

$$\text{PF} = \frac{\text{Total \# of cells}}{\text{\# of participating cells}}$$

The PF before partitioning and binning is shown in column 4. This corresponds to the PF of a monolithic implementation. Note that this can be as high as ∼8,300. The PF after partitioning and binning is shown in column 5, and it is about 10 on average. Attempting to lower the PF beyond this value results in several variables appearing in multiple banks. The total number of strips for all the examples is shown in column 6. Note that all examples require less than 12,500 strips, indicating that they would fit on our IC. This is a dramatic improvement in capacity over existing monolithic hardware-based SAT approaches, which can handle between 1,280 and 24,700 clauses with 64 FPGA boards or 121 configurable processors, respectively, as opposed to about 63,000 clauses on a *single IC* for our approach. Further, the total runtime for the partitioning (using hMetis [17]), diagonalization, and greedy bin packing for the examples listed in Table 4.3 ranged from 8 to 200 s on a 3.6 GHz, 3 GB machine running Linux. These runtimes are significantly lower than the BCP-based software SAT runtimes for these examples. Even if the partitioning runtimes were higher, the time spent in partitioning is amply recovered when multiple SAT calls need to be made for the same instance.

The delay of each bank (the difference between the time a new decision variable is driven to the time the last implication is driven out by the bank) was computed via SPICE simulations to be $\Delta_B = 3$ ns (for a bank with 3 strips, which is approximately the average number of strips per clause as indicated in column 7 of Table 4.3). We also estimated the delay due to the inter-bank communication via SPICE simulations. To do this, we first found the average number of implications caused by any decision, over all the examples under consideration. The average number of impli-

cations per decision was found to be about 21. For the computation of delay due to inter-bank communication, we conservatively assumed that the average number of implications per decision was 25. We assumed the worst-case situation (where each of these 25 implications is on variables that repeat across banks, with a repeat level of 0). This results in the slowest inter-bank communication scenario. Using SPICE delay values (computed using layout-extracted wiring parasitics), we obtained the values of the delay between communication units at level i and $i + 1$. Let this delay be denoted by Δ_i. Then the total delay is estimated as

$$\Delta_C = 2 \cdot 25 \cdot \Sigma_{i=0}^{5}(\Delta_i) + \Delta_B$$

Note that long wires (between communication units at different repeat levels) are optimally buffered for minimal delay. Using the values of Δ_i that we obtained, Δ_C is computed to be 27 ns. Using this estimate, we compute the time for the solving of the SAT problem in our hardware SAT engine as

Our Runtime = Number of Decisions $\cdot \Delta_C$

The worst-case time to generate and communicate implications (Δ_C) dominates the conflict analysis time, and hence our runtime estimates are based on Δ_C alone. Our runtime is compared, in Table 4.4, against MiniSAT[2], a state-of-the-art BCP-based software SAT solver. We modified MiniSAT in two ways, in order to estimate the runtime of our hardware approach. First, we modified MiniSAT to implement a static decision strategy which is the same as the decision strategy used in our hardware engine. MiniSAT performs a smart conflict clause simplification by applying subsumption resolution [36] and caching of intermediate results. So, in our second modification of MiniSAT, we disabled any simplification of the conflict clauses. This variant of MiniSAT (modified in the above two ways) is referred to as MiniSAT* in the sequel. The number of decisions made by MiniSAT* was used in computing our runtime using the above equation. Columns 2 and 3 of Table 4.4 list the number of decisions and the number of conflicts reported by MiniSAT. Column 4 lists the MiniSAT runtimes. The MiniSAT runtimes for these instances were obtained on a 3.6 GHz, 3 GB machine running Linux. Columns 5 and 6 list the number of decisions and the number of conflicts reported by MiniSAT*. Our estimated runtimes are reported in column 7. The speedup obtained over MiniSAT is reported in column 8. The average speedup over MiniSAT obtained is 1.84×10^3.

In other words, our approach yields over 3 orders of magnitude improvement in runtime over an advanced BCP-based software SAT solver. It achieves 1–2 orders of magnitude speedup over other hardware SAT approaches as well. Other hardware SAT approaches have significant capacity problems, making them impractical for large instances. Our approach has a large capacity and is highly scalable, and hence is ideally suited for large SAT instances.

In order to estimate the power consumption of our approach, we conducted additional SPICE simulations. These simulations were performed for computing the average power required for a single implication within a bank and the average power required for communicating this implication to every other bank. The power consumption for the long wires (between communication units at different repeat

Table 4.4 Comparing against MiniSAT (a BCP-based software SAT solver)

	MiniSAT			MiniSAT*			
Instance	# Decisions	# Conflicts	MiniSAT runtime (s)	# Decisions	# Conflicts	Our Runtime(s)	Speed Up
par16-3	6.26×10^{3}	5.98×10^{3}	5.68×10^{-1}	1.43×10^{4}	1.15×10^{4}	3.11×10^{-4}	1.83×10^{3}
ii8b4	5.70×10^{2}	0	6.00×10^{-3}	5.01×10^{2}	0	1.35×10^{-5}	4.44×10^{2}
am	4.64×10^{7}	3.95×10^{7}	1.26×10^{4}	4.62×10^{9}	3.64×10^{9}	1.24×10^{2}	1.02×10^{2}
par32-5	6.62×10^{7}	6.14×10^{7}	5.36×10^{3}	5.53×10^{8}	4.25×10^{8}	1.49×10^{1}	3.60×10^{2}
ii16a1	9.07×10^{2}	7	1.30×10^{-2}	9.70×10^{2}	3	2.03×10^{-5}	6.40×10^{2}
ii32c4	4.50×10^{1}	4	1.90×10^{-2}	1.50×10^{2}	9.90×10^{1}	3.15×10^{-6}	6.03×10^{3}
dekker	6.89×10^{5}	5.87×10^{5}	5.35×10^{2}	3.81×10^{6}	1.83×10^{6}	1.03×10^{-1}	5.19×10^{3}
frg2mul	3.24×10^{6}	6.07×10^{5}	6.21×10^{2}	1.57×10^{8}	2.09×10^{7}	4.24	1.47×10^{2}
AVG							1.84×10^{3}

levels) for the latter experiment was computed using layout-extracted wiring parasitics. The value obtained was $P_{\text{single}}^{\text{comm.}} = \sim 3.69$ nW. Again assuming the worst-case situation (where each of the 25 implications/decision is on variables that repeat across banks, with a repeat level of 0), the total power required for all communications per decision (per clock cycle) is

$$P_{\text{comm.}} = P_{\text{single}}^{\text{comm.}} \cdot 25 = 92.25\,\text{nW}$$

The average power consumed by the clause bank for generating an implication, $P_{\text{single}}^{\text{imp}}$, was obtained to be about 0.363 μW. The total number of banks per IC would be at most 64 (since only 6 levels of hierarchy are present in the IC). In the worst case, assume that the partitions obtained from hMetis repeat a single variable v over all the 64 banks. Now suppose that there is an implication on v in every bank. For driving an implication, as explained in the previous sections, only one of the *lit* or *lit_bar* signal along with the *var_implied* signal is driven. For a conflict, on the other hand, all three signals are driven. Therefore the average power consumption for driving a single conflict literal ($P_{\text{single}}^{\text{conf}}$) is (3/2) $\cdot P_{\text{single}}^{\text{imp}}$. Since there are on average 25 implications per decision, and assuming each decision leads to a conflict involving each of the 25 implications, there are in the worst case 25 implied variables that can participate in analyzing the conflict. Hence the average power for the BCP engine (which performs implication/conflict analysis) per clock cycle is

$$P_{\text{BCP}} = P_{\text{single}}^{\text{conf}} \cdot 25 \cdot \text{Number of Banks} = 871.2\,\mu\text{W}$$

The worst-case power per cycle for our hardware SAT solver is therefore

$$P_{\text{avg}} = P_{\text{BCP}} + P_{\text{comm.}} = 871.3\,\mu\text{W}$$

Note that this low power arises from the fact that in practice, there is very little conflict activity whenever any decision is made. A majority of the clause cells do not participate in a conflict, thereby keeping the worst-case power consumption low.

For the examples listed in Table 4.3 we compared the BCP-based software SAT runtimes with or without a limit on the number and width of the conflict clauses. The purpose of this experiment was to determine if limiting the number and width of conflict clauses significantly affects SAT runtimes. The number and width of clauses corresponded to a single row of clause banks in the center of the chip. With this limit, we noted a negligible difference in the SAT runtimes compared to the case when there was no limit (for a timeout of 1 h). Since our clause banks can be interchangeably used for conflict clause storage and regular clause storage, we can handle larger SAT instances by storing fewer conflict clauses in the IC.

Larger designs can be handled elegantly by our approach, since multiple SAT ICs can be connected to work cooperatively on a single large instance. A pair of such ICs would effectively implement an additional level in the inter-bank communication tree. The only wires that are shared between two such ICs are those implementing

inter-bank communication. By implementing these using fast board-level IO, the system of cooperating SAT ICs can be made to operate extremely fast. The decision engine of each IC other than the root IC behaves as a communication unit, in such a scenario.

4.9 Chapter Summary

In this chapter, we have presented a custom IC implementation of a hardware SAT solver and also augmented it for extracting the minimum unsatisfiable core. The speed and capacity for our SAT solver obtained are dramatically higher than those reported for existing hardware SAT engines. The speedup comes from performing the tasks of computing implications and determining conflicts in parallel, using a specially designed clause cell. Approaches to partition a SAT instance into banks and bin them into strips have been developed, resulting in a very high utilization of clause cells. Also, through SPICE simulations we determined that the average power consumed per cycle by our SAT solver is *under 1 mW* which further strengthens the practicality of our approach. Note that although we used a variant of the BCP engine of GRASP [28] in our hardware SAT solver, the hardware approach can be modified to implement other BCP engines as well. For extracting the unsatisfiable core, we implemented the approach described in [19] since our architecture naturally complements the technique proposed in [19]. Also the additional optimizations of [19] can be seamlessly implemented in our architecture.

References

1. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/. The DIMACS ftp site
2. http://www.cs.chalmers.se/cs/research/formalmethods/minisat/main.html. The MiniSAT Page
3. http://www.lri.fr/∼simon/contest04/results/. The SAT'04 Competition
4. Abramovici, M., Saab, D.: Satisfiability on reconfigurable hardware. In: Proceedings, International Workshop on Field Programmable Logic and Applications, pp. 448–456 (1997)
5. Abramovici, M., de Sousa, J., Saab, D.: A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In: Proceedings, Design Automation Conference (DAC), pp. 684–690 (1999)
6. Bruni, R., Sassano, A.: Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In: LICS Workshop on Theory and Applications of Satisfiability Testing (2001)
7. Büning, H.K.: On subclasses of minimal unsatisfiable formulas. Discrete Applied Mathematics **107**(1–3), 83–98 (2000)
8. Cook, S.: The complexity of theorem-proving procedures. In: Proceedings, Third ACM Symposium on Theory of Computing, pp. 151–158 (1971)
9. Davydov, G., Davydova, I., BÂÎuning, H.K.: An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF. In: Annals of Mathematics and Artificial Intelligence, vol. 23, pp. 229 – 245 (1998)
10. Fleischner, H., Kullmann, O., Szeider, S.: Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. In: Theoretical Computer Science, vol. 289, pp. 503–516 (2002)

11. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Proceedings, Design, Automation and Test in Europe (DATE) Conference, pp. 142–149 (2002)
12. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proceedings, Design and Test in Europe Conference, pp. 10,886 – 10,891 (2003)
13. Gu, J., Purdom, P., Franco, J., Wah, B.: Algorithms for the satisfiability (SAT) problem : A survey. DIMACS Series in Discrete Mathematics and Theoretical Computer Science **35**, 19–151 (1997)
14. Gulati, K., Waghmode, M., Khatri, S., Shi, W.: Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction. IET Computers Digital Techniques **2**(3), 214–229 (2008)
15. Huang, J.: MUP: A minimal unsatisfiability prover. In: ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference, pp. 432–437 (2005)
16. Jin, H., Awedh, M., Somenzi, F.: CirCUs: A satisfiability solver geared towards bounded model checking. In: Computer Aided Verification, pp. 519–522 (2004)
17. Karypis, G., Kumar, V.: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices. http://www-users.cs.umn.edu/∼karypis/metis (1998)
18. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Proceedings, Tenth European Conference on Artificial Intelligence, pp. 359–363 (1992)
19. Lynce, J., Marques-Silva, J.: On computing minimum unsatisfiable cores. In: In Seventh International Conference on Theory and Applications of Satisfiability Testing, pp. 305–310 (2004)
20. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proceedings, Computer Aided Verification, pp. 1–13 (2003)
21. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the Design Automation Conference, pp. 530–535 (2001)
22. Nagel, L.: SPICE: A computer program to simulate computer circuits. In: University of California, Berkeley UCB/ERL Memo M520 (1995)
23. Nam, G.J., Sakallah, K.A., Rutenbar, R.A.: Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based Boolean SAT. In: FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, pp. 167–175 (1999)
24. Oh, Y., Mneimneh, M., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: A minimally unsatisfiable subformula extractor. In: Proceedings, Design Automation Conference, pp. 518–523 (2004)
25. Pagarani, T., Kocan, F., Saab, D., Abraham, J.: Parallel and scalable architecture for solving Satisfiability on reconfigurable FPGA. In: Proceedings, IEEE Custom Integrated Circuits Conference (CICC), pp. 147–150 (2000)
26. Papadimitriou, C.H., Wolfe, D.: The complexity of facets resolved. In: Journal of Computer and System Sciences, pp. 37(1):2–13 (1998)
27. Reis, N.A., de Sousa, J.T.: On implementing a configware/software SAT solver. In: FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, p. 282. IEEE Computer Society, Washington, DC (2002)
28. Silva, M., Sakallah, J.: GRASP-a new search algorithm for satisfiability. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 220–7 (1996)
29. Skliarova, I., Ferrari, A.: Reconfigurable hardware SAT solvers: A survey of systems. IEEE Transactions on Computers **53**(11), 1449–1461 (2004)
30. de Souza, J.T., Abramovici, M., da Silva, J.M.: A configware/software approach to SAT solving. In: IEEE Symposium on FPGAs for Custom Computing Machines (2001)
31. Suyama, T., Yokoo, M., Sawada, H., Nagoya, A.: Solving satisfiability problems using reconfigurable computing. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9**(1), 109–116 (2001)

32. Waghmode, M., Gulati, K., Khatri, S., Shi, W.: An efficient, scalable hardware engine for Boolean satisfiability. In: Proceedings, IEEE International Conference on Computer Design (ICCD), pp. 326–331 (2006)
33. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, p. 10880. IEEE Computer Society, Washington, DC (2003)
34. Zhao, Y., Malik, S., Moskewicz, M., Madigan, C.: Accelerating Boolean satisfiability through application specific processing. In: Proceedings, International Symposium on System Synthesis (ISSS), pp. 244–249 (2001)
35. Zhao, Y., Malik, S., Wang, A., Moskewicz, M., Madigan, C.: Matching architecture to application via configurable processors: A case study with Boolean satisfiability problem. In: Proceedings, International Conference on Computer Design (ICCD), pp. 447–452 (2001)
36. Zheng, L., Stuckey, P.J.: Improving SAT using 2SAT. In: ACSC '02: Proceedings of the Twenty-fifth Australasian Conference on Computer Science, pp. 331–340. Australian Computer Society, Inc., Darlinghurst, Australia (2002)
37. Zhong, P., Ashar, P., Malik, S., Martonosi, M.: Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability. In: Proceedings, Design Automation Conference, pp. 194–199 (1998)
38. Zhong, P., Martonosi, M., Ashar, P., Malik, S.: Accelerating Boolean satisfiability with configurable hardware. In: Proceedings, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 186–195 (1998)

Chapter 5
Accelerating Boolean Satisfiability on an FPGA

5.1 Chapter Overview

In this chapter, we propose an FPGA-based SAT approach in which the traversal of the implication graph as well as conflict clause generation is performed in hardware, in parallel. In our approach, clause literals are stored in the FPGA slices. In order to solve large SAT instances, we heuristically partition the clauses into a number of 'bins,' each of which can fit in the FPGA. This is done in a preprocessing step. These bins may share variables and hence are not independent sub-problems. The FPGA operates on one bin at a given instant, and the FPGA hardware also coordinates the handling of the bins of the entire instance. An on-chip block RAM (BRAM) is used for storing all the bins (or caching a portion of the bins) of a partitioned CNF problem. The embedded PowerPC processor on the FPGA performs the task of loading the appropriate bin from the BRAM. The core routines of conflict clause generation and Boolean constant propagation (BCP) are performed in parallel in the hardware (implemented in Verilog). The entire flow, which includes the preprocessing step, loading the BRAM, programming the PowerPC, and the subsequent communications between partitions (which is required for BCP, conflict clause generation, and non-chronological backtracking (both inter- and intra-bin)), has been automated and verified for correctness using a Virtex-II Pro (XC2VP30) FPGA board. The experimental results and their analysis, along with the performance models derived from these results, are discussed in detail. Further, we show that an order of magnitude improvement in runtime can be obtained over MiniSAT (the best-in-class software-based approach) by using a Virtex-4 (XC4VFX140) FPGA device. The resulting system can handle instances with as many as 10K variables and 280K clauses.

The rest of this chapter is organized as follows. The motivation for this work is described in Section 5.2. Section 5.3 discusses previous FPGA-based SAT solvers. Section 5.4 describes the hardware architecture employed in our approach. A general flow for solving a CNF instance which is partitioned into bins is described in Section 5.5. Section 5.6 describes the up-front clause partitioning methodology, which targets maximum utilization of the hardware with low variable overlap. The hardware details for our approach are explained in Section 5.7. Section 5.8 reports

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_5,
© Springer Science+Business Media, LLC 2010

our current implementation on a low-end FPGA evaluation board, followed by projected performance numbers on a high-end FPGA board. These projections are derived based on a performance model extracted from detailed performance data from our current system. Section 5.9 summarizes the chapter.

5.2 Introduction

As mentioned in the last chapter, Boolean satisfiability (SAT) [3] is a core NP-complete problem, and hence there is a strong motivation to accelerate SAT. In this work, we propose an FPGA-based approach to accelerate the SAT solution process, with the goal of speedily solving large instances in a *scalable* fashion. By scalable, we mean that the same platform can be easily made to work on larger SAT instances. The FPGA-based hardware implements the GRASP [11] strategy of non-chronological backtracking. In our approach, a predetermined number of clauses of fixed width are implemented on the FPGA. The SAT problem is mapped to this architecture in an initial step which partitions the original SAT instance into *bins* which can be solved on the FPGA. Further, inter-bin (as well as intra-bin) non-chronological backtrack is implemented in our approach. Our hardware approach performs, in parallel, both the tasks of implicit traversal of the implication graph and conflict clause generation. The contribution of this work is to come up with a high capacity, fast, scalable FPGA-based SAT approach. We do not claim to propose any new SAT solution heuristics in this work. Similar to our custom IC-based SAT solver described in the last chapter, we have used the GRASP [11] engine in our FPGA-based SAT solver. As before, the hardware approach can be modified to implement other BCP engines, since the BCP logic of any BCP-based SAT solver can be ported to an HDL and directly synthesized in our approach.

Our approach is implemented and tested on a Xilinx Virtex-II Pro evaluation board. Experimental results on LUT utilization and performance figures are derived from an actual implementation using an XC2VP30 Virtex-II Pro based FPGA platform. The results from these experiments are projected to an industrial strength FPGA system and indicate a 17× speedup over the best-in-class software approach. The resulting system can handle instances with as many as 10K variables and 280K clauses.

5.3 Previous Work

In addition to the existing work discussed in the previous chapter, several FPGA-based SAT solvers have been reported in the past. We classify them into *instance-specific* and *application-specific* approaches. In instance-specific approaches the hardware is recompiled for every instance. This is a key limitation, since compilation times for an FPGA can take several hours. The speedup numbers reported

in the instance-specific approaches, however, do not present the compilation and configuration times. Approaches which are not instance specific are *application specific*.

Instance-specific approaches reported in literature are [14, 6, 13, 2, 7]. Among these approaches, as reported in [6], the largest example that can be handled has about 1,300 clauses with an average speedup of 10×. Our approach, in contrast, is application specific and thus the same device, once configured, can be used multiple times for different instances. Further, our approach can obtain a 17× speedup over the best-in-class software approach, with a capacity of 10K variables and 280K clauses.

The multi-FPGA approach described in [15] demonstrates non-chronological backtracks and dynamic addition of conflict-induced clauses. However, the approach is instance specific and requires re-synthesis, remapping, and regeneration and reconfiguration of the bit stream, each time a conflict-induced clause is added to the clause database. The approach claims to perform these repeated tasks in an incremental fashion which is possible due to a regular hardware structure. The new compile times (obtained with the incremental tasks) are a few orders of magnitude higher than the actual runtime for most instances reported in their results. Our approach, in contrast, uses a *single FPGA device*. For problem instances which cannot be accommodated in a monolithic fashion in a single FPGA, we partition the instance into 'bins' of clauses (which may share common variables). *This allows our approach to scale elegantly and solve large SAT problems, unlike previous reconfigurable approaches*. Each partition is then solved for satisfiability, while maintaining consistency with the existing global decisions and assignments. This may require backtracking to a previous bin. Backtracking in our approach is performed in a non-chronological fashion, even across bins. No other existing application-specific hardware or reconfigurable SAT solver exhibits a non-chronological backtrack and dynamic addition of conflict-induced clauses, carried out entirely in hardware.

All existing application-specific hardware or reconfigurable approaches eventually run into the problem of an instance not fitting in a single FPGA or reconfigurable device. The approach of [5, 7] implements the prototype on a Pamette board containing four Xilinx XC4028 FPGAs. These approaches do not propose anything for solving problem instances whose size exceeds the board capacity. The application-specific approach in [8], like [13, 16], employs several interlinked FPGAs, but assumes that the FPGA resources are sufficient for solving a SAT instance. Also, the runtimes they reported were achieved based on software simulations.

There are some application-specific approaches which can handle instances that do not fit in a single FPGA device. The approaches described in [9, 10], like our approach, are implemented on a single FPGA board. However, in these approaches, the memory module storing the instance has to be reconfigured for different problem instances (or independent sub-instances for large instances). Their authors do not clarify the procedure followed when independent sub-instances of feasible sizes cannot be obtained. The consistency of assignments across sub-instances is not trivial to maintain in hardware, but this is not addressed. *Our approach maintains this*

consistency, and backtracks to a previous partition (bin) non-chronologically, in case the offending decision was not made in the current partition. The approach of [12] creates a matrix (where rows are clauses and column are variables) from the problem instance and searches for a ternary vector orthogonal to every row, in order to satisfy the instance. For larger instances, it attempts at solving the problem in software, until the sub-instance size is accommodable in the FPGA. In our approach, software is used only for the initial partitioning and clause transfer; thereafter, all steps are performed entirely in hardware. Further, the speedups reported in this paper against GRASP are nominal and only for the *holex* benchmarks. Our approach reports speedup against MiniSAT [1], which is known to be significantly faster than GRASP. Our results are presented over a variety of benchmarks.

The work presented in this chapter is an FPGA version of the custom IC-based SAT solver described in the previous chapter. However, the custom IC approach solves the entire instance in a monolithic fashion. Our FPGA-based approach, on the other hand, partitions a CNF instance into bins and is required to maintain consistency in assignments across all bins while solving one bin at a time. This requires several changes in the preprocessing step, the hardware design, and the overall flow. An extended abstract of our FPGA-based SAT solver is described in [4].

5.4 Hardware Architecture

5.4.1 Architecture Overview

Figure 5.1 shows the hardware architecture of our application-specific (i.e., not instance-specific) approach. We use an FPGA board that has a duplex communication link with the host system. The FPGA is first loaded with the configuration information for our SAT engine. No instance information is loaded at this stage. Since most practical-sized CNF instances would not readily fit on the FPGA fabric, we heuristically partition the original CNF into smaller CNFs, called *bins*, such that their inter-dependence is reduced. In other words, we aim at reducing the number of common variables across bins. Also, each of these bins is sized such that the bin can individually fit in the FPGA fabric. This partitioning is performed as a preprocessing step on the host system, before loading the bins to the FPGA board. In reality, multiple CNF instances (each in its respective partitioned 'bin' format) are stored in a 512 MB DDR DRAM memory card which is on the FPGA board. These partitioned CNF instances are first loaded onto the on-board DRAM from the host system using board-level I/O. Next, all the bins of one of these CNF instances are loaded in the on-chip block RAM (BRAM). This is the instance which is being currently processed, and we refer to it as the *current instance* in the sequel. Note that bins can potentially be cached in the BRAM, enhancing scalability. The FPGA is then loaded with one of the bins of the current instance. This is done using an embedded PowerPC processor, which transfers the bin data from the BRAM to the FPGA fabric. The on-chip PowerPC manages both the loading of the current

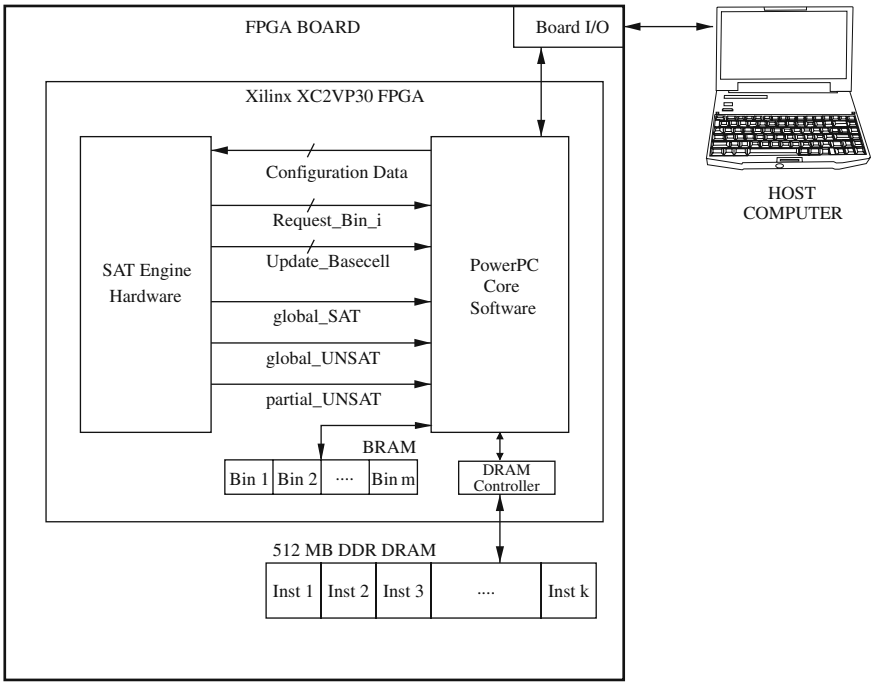


Fig. 5.1 Hardware architecture

instance from the DRAM to the BRAM and the loading/unloading of bins from BRAM onto the FPGA (as dictated by the hardware). These transfers are performed using bus transfer protocol IPs provided by Xilinx. These IPs allow transfers across the processor local bus (PLB) and on-chip peripheral bus (OPB). After the bin is loaded into the FPGA fabric, the FPGA starts to perform implication and conflict clause generation in parallel. The next section discusses our approach of solving a CNF instance (which is partitioned across several bins) in our FPGA-based hardware SAT solver.

5.5 Solving a CNF Instance Which Is Partitioned into Several Bins

As mentioned above, the original CNF instance C is initially partitioned into smaller bins, $b_1, b_2, \ldots, b_n$. Our hardware engine tries to satisfy each bin b_i, using the stored global assignments on the variables. In this section our flow for solving a partitioned SAT instance is explained. Implementation details are given in the sequel.

The variables V of the CNF C are statically awarded a decision level once the bins have been created. Along with each bin b_i, we load the decision levels of the variables $V_i \subseteq V$ it contains, along with the current *state* of every variable $v \in V_i$.

The global state of all variables V is stored in the on-chip BRAM. The state of a variable consists of the following information:

- whether the variable has been decided (assigned or implied);
- the current decision on the variable;
- if the variable has been decided, the decision level it was decided at;
- if the variable has been decided, the bin it was decided in; and
- if the decision on this variable is the highest decision level we backtracked on.

We begin by solving the first bin. After any bin is solved, it results in a *partial_SAT* or *partial_UNSAT* condition. *Partial_SAT* indicates that the current bin has all clauses satisfied, with no conflicts with the current status of any variable in V. A *partial_UNSAT* indicates the opposite. If a bin b_i is *partial_SAT*, we first update the states of the variables $v \in V_i$ into the global state. Also, any learned clauses generated during the operation on b_i are appended to the clauses of bin b_i in the BRAM. We then load the FPGA with the clauses of bin b_{i+1} and the states of the variables $v \in V_{i+1}$. The SAT engine then attempts to *partial_SAT* this new bin. With every *partial_SAT* outcome on bin j, we proceed in a sequential manner from bin b_j to b_{j+1}. If the last bin is also *partial_SAT*, we declare the instance to be *global_SAT* or *satisfiable*.

In case there is a conflict with an existing state of variables $\{v_c\}$, we non-chronologically backtrack on v_{bkt}, which is the variable with the highest decision level among $\{v_c\}$. If the variable v_{bkt} was assigned in the current bin itself, we simply revert our decision. If the variable v_{bkt} was implied in the current bin, we backtrack on the variable which caused the implication. On the other hand, if the variable v_{bkt} was assigned or implied in a previous bin, we declare the current bin to be *partial_UNSAT*. Using the information contained in the state of this variable, we obtain the bin number we need to backtrack to, in order to revert the decision on v_{bkt}. This allows us to *backtrack across* bins. In other words, we perform non-chronological backtrack *within* bins and also *across* bins, ensuring the completeness of our SAT procedure. Let the new bin be b_j. Now we load the FPGA with the clauses of b_j and the states of the related variables $v \in V_j$. On reverting the decision on v_{bkt} (which could require recursive backtracking), we delete the decisions on all variables with a decision level higher than v_{bkt}'s decision level. We then continue as usual with the updated state of variables. As before, if bin b_j is now *partial_SAT*, we next load the FPGA with bin b_{j+1}. During conflict analysis, if the earliest decision level has been backtracked on, and the current status of the variables still leads to a conflict, we declare the instance to be *global_UNSAT* or *unsatisfiable*.

In our approach, the FPGA hardware performs the satisfiability check of a bin, as well as non-chronological (inter and intra-bin) backtrack. The software (running on the embedded PowerPC) simply loads the next bin as requested by the hardware.

Each time a bin is loaded onto the FPGA, we say that the bin has been 'touched,' in the sequel. The flow explained above allows us to perform BCP and non-chronological backtrack. The next section details the algorithm used for partitioning the CNF instance across bins.

5.6 Partitioning the CNF Instance

To partition a given CNF instance into multiple bins of bounded size (which can fit in the FPGA fabric) we use a two-dimensional graph bandwidth minimization algorithm, followed by greedy bin packing. Let us view the CNF instance as a matrix whose columns are labeled as variables and rows as clauses. The bandwidth minimization algorithm attempts to diagonalize this matrix. For each clause C_i, we assign it a *gravity* $G(C_i)$ which is computed as follows: $G(C_i) = \sum_{C_j \in R(C_i)} (P(C_j) \cdot S(C_i,C_j))$.

Here, $R(C_i)$ is the set of clauses which have at least one variable common with clause C_i and $P(C_j)$ is the index of the current row of C_j and $S(C_i,C_j)$ is the number of common variables between clauses C_i and C_j.

The exact dual is used for computing the gravity of every variable in the CNF instance. The pseudocode for the bandwidth minimization algorithm is shown in Algorithm 3.

Algorithm 3 Pseudocode for Bandwidth Minimization

```
Best_Cost = Infinity
for i = 1; i ≤ Number of iterations; i++ do
    Compute Gravity of all clauses
    Rearrange Clauses in increasing order of gravity
    Compute Gravity of all variables
    Rearrange Variables in increasing order of gravity
    Greedy Bin packing for creating Bins
    Compute cost of current arrangement Cost_i
    if (Best_Cost ≥ Cost_i) then
        Best_Cost = Cost_i
        Store current arrangement
    end if
end for
return(Stored Arrangement)
```

As shown in Algorithm 3, we alternate the gravity computation and rearrangement between clauses and variables. With every rearrangement of clauses and variables in an increasing order of gravity, we compute a new cost. The cost of the arrangement is the equally weighted sum of the following:

- Number of bins. A smaller number of bins would reduce the overhead involved with loading the FPGA with a new bin and also reduce communication while solving the instance.
- The sum, across all variables v in the CNF instance, of the number of bins in which v occurs. The intuition for this cost criterion is to reduce the overlap of variables across bins. A larger overlap would require more consistency checks and possibly more backtracks across bins.
- The sum across all variables v in the CNF instance, of the number of bins v *spans*. By *span* we mean the difference between the largest and the smallest bin index, in which v occurs. While backtracking, we delete the intermediate decisions and

variables. Therefore, this criterion would help us reduce the amount of data deletion which may be possibly done during backtracks.

The greedy bin packing step simply packs the rearranged CNF instance into bins which have a predetermined maximum number of clauses $C_{\max}$ and variables $V_{\max}$ (such that any bin can fit monolithically in the FPGA fabric). We take $k \leq C_{\max}$ clauses and assign them to a new bin provided the variable support of these clauses is less than or equal to $V_{\max}$.

The hardware details of our implementation are discussed in the next section.

5.7 Hardware Details

Our FPGA-based SAT solver is based partly on the custom IC approach presented in Chapter 4. Hence, the reader is referred to the previous chapter for some details of the hardware. In particular, the abstract view of our SAT solver for a single bin is identical to the abstract view of the monolithic 'clause bank' described in the last chapter. Also, the clause cell and its implementation for generating implications and conflict-induced clauses for a single bin is identical to the clause cell described in the previous chapter. The only differences between the clause bank in the last chapter and the single bin in the current chapter are as follows:

- There is no precharge logic in the FPGA-based approach.
- There are no wired-OR signals in the FPGA-based approach.
- Each bidirectional signal in the clause cell described in Chapter 4 is replaced by a pair of unidirectional *in* (input) and *out* (output) signals.
- There is no termination cell in the FPGA approach. This type of cell was used to allow more than one clause to reside on the same row of the clause bank in Chapter 4.
- In the FPGA approach, the learned clauses for bin b_i are updated into the bin b_i. In Chapter 4, learned clauses were simply added to the clause bank. However, in the FPGA-based approach, in order for a subsequent load of some bin i to take advantage of a previously computed conflict-induced clause for that bin, these learned clauses are added to the clause database of bin i in the BRAM.

The decision engine state machine in the current FPGA-based approach is enhanced in order to process a CNF instance in a partitioned fashion. This is discussed next.

Figure 5.2 shows the state machine of the decision engine. To begin with, the first bin of the current CNF instance is loaded onto the hardware. All signals are initialized to their *refresh* state. The decision engine assigns the variables in the order of their *identification tag*, which is a numerical ID for each variable, statically assigned such that most commonly occurring variables are assigned a lower tag. The decision engine assigns a variable (in the *assign_next_variable* state) and this assignment is forwarded to all the clauses of the bin. The decision engine then waits for the bin to compute all the implications during the *wait_for_implications* state.

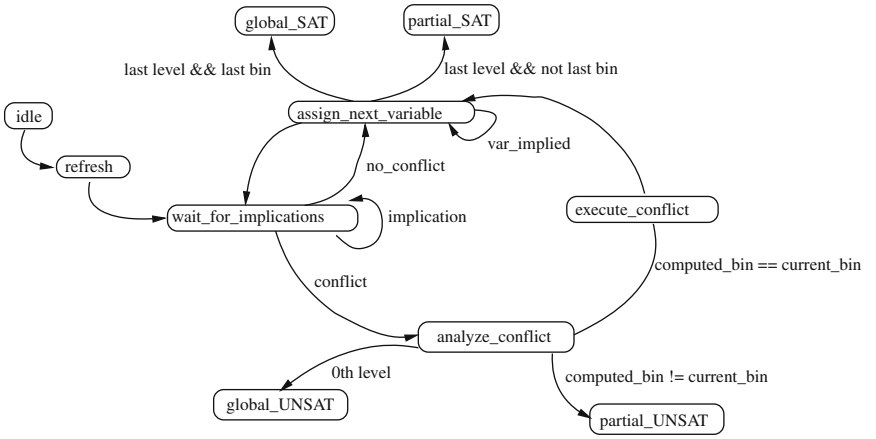


Fig. 5.2 State diagram of the decision engine

For bins other than the first bin, the decision engine at first just propagates any existing decisions on any of the variables of this bin, in the ascending order of their decision levels. All variables implied due to these existing assignments store the decision level of the existing assignment due to which they were implied. Similarly, all variables implied due to a new assignment store the decision level of the newly assigned variable as their decision level. All implied variables store the current bin number in their state information.

When an assignment is made, if no conflict is generated due to the assignment, the decision engine assigns the next unassigned variable in the current bin. If the next unassigned variable v does not occur in any of the clauses of the current bin, or all clauses containing v are already satisfied, the decision engine skips an assignment on this variable and proceeds to the next variable. This helps in avoiding an unnecessary decision on a variable which could lead to a backtrack from another bin in the future. If all the clauses of the current bin b_i are satisfied and there are no conflicts, then b_i is declared to be *partial_SAT*. A new bin, b_{i+1}, is loaded on to the FPGA along with the states of its related variables. If the last bin is *partial_SAT*, the given CNF instance is declared to be *global_SAT* or *satisfiable*.

If there is a conflict in b_i, all the variables participating in the conflict clause are communicated by the clauses in the bin, to the decision engine. Based on this information, during the *analyze_conflict* state, the conflict-induced clause is generated and stored in the FPGA fabric, just like regular clauses. Also the decision engine non-chronologically backtracks according to the GRASP [11] algorithm. Using the information contained in the state of a variable, the engine can compute the latest assignment among the variables participating in the conflict and the bin (*backtrack* bin) where the assignment on this variable was made. When the backtrack bin is the current bin, and the backtrack level is lower than a variable's stored decision level, then the stored decision level is cleared before further action by the decision engine during the *execute_conflict* state. When the backtrack bin is not the current bin,

the decision engine goes to the *partial_UNSAT* state, causing the required bin to be loaded. After a conflict is analyzed, the backtracked decision is applied. The variable to be backtracked on is flagged with this information. At any given instance, only the flag of the lowest indexed variable is recorded. If a backtrack has been requested on every variable involved in a conflict, and a conflict exists even by backtracking on the earliest decision, the given CNF is declared as *global_UNSAT* or *unsatisfiable*.

Our FPGA-based SAT solver is a GRASP [11] based algorithm with static selection of decision variables. Just like GRASP, it performs non-chronological backtracking and dynamic addition of conflict-induced clauses. As a result, it retains (within as well as across bins) the completeness property of GRASP.

5.8 Experimental Results

The experimental results are discussed in the following sections. Section 5.8.1 discusses our current implementation briefly. Our working system is implemented on an FPGA evaluation board. In order to obtain projected performance numbers on a high-end FPGA board, we first extract detailed performance data from our system. Using this data, we develop a mathematical performance model, in Section 5.8.2, which estimates the bin size, numbers of bins touched, and communication speeds as a function of SAT problem. Using this performance model, we project the system performance (using our existing performance data) for industrial-strength FPGA boards, in Section 5.8.3.

5.8.1 Current Implementation

To validate our approach, we have implemented our hardware SAT solver on a Xilinx XC2VP30 device-based evaluation board using ISE 8.2i for hardware (Verilog) and EDK 8.2i for instantiating the PowerPC, processor local bus (PLB), on-chip peripheral bus (OPB), BRAM, and PLB2OPB bridge cores. Our current implementation can solve CNF instances of size 8K variables and 31K clauses. If we were to cache the bins in BRAM, then the capacity of the system increases to 77K clauses over 8K variables. The size of a single bin is 16 variables and 24 clauses. Of these, four clauses are designated as learned clauses. The FPGA device utilization with this configuration, including the EDK cores, is $\sim$70%. With larger FPGAs, significantly larger CNFs can be tackled.

Our current implementation correctly solves several non-trivial CNF instances. Our regression suite consists of about 10,000 satisfiable and unsatisfiable instances. To validate intermediate assignments and decisions at the bin level, we performed aggressive testing. Each partial bin assignment is verified against MiniSAT [1]. This is done as follows: Say the mth bin is part_SAT and let the set of current assignments be p_m. A CNF instance C is created which includes all clauses from bins 1 through m and single literal clauses using the current assignments, i.e., set p_m:

$C = [\prod_{i=1}^{m} (bin_i)] \cdot p_m$
The CNF instance, C, thus generated is solved using MiniSAT and verified to be satisfiable. Similarly, if the nth bin is part_UNSAT, a CNF instance D, s.t.
$D = [\prod_{i=1}^{n} (bin_i)] \cdot p_n$
is generated and solved using MiniSAT. This should be unsatisfiable. Several of our regression instances touch thousands of bins and the assignments until each bin is verified in this fashion. As mentioned previously, several CNF instances, after being partitioned into bins, are loaded onto the board DRAM. Typically hundreds of such instances are loaded at a time and tested for satisfiability one after another. Only the current instance resides completely in the on-chip BRAM.

5.8.2 Performance Model

5.8.2.1 FPGA Resources

We conducted several FPGA synthesis runs, using different bin sizes, to obtain the dependence of FPGA resource utilization on bin size. The aim of this experiment was to quantify the FPGA resource utilization as a function of

- number of variables of the bin and
- number of clauses in the bin.

Based on several experiments, we conclude that the LUT utilization is $20 \cdot V \cdot C + 300 \cdot V$, where V and C are the number of variables and the number of clauses per bin, respectively. Figures 5.3 and 5.4 graphically show the increase in number of LUTs used, with an increase in number of clauses and an increase in number of variables, respectively. The X-axis of Fig. 5.3 represents the number of clauses in a single bin which is currently stored in the FPGA fabric. The X-axis of Fig. 5.4 represents the number of variables in a single bin, configured onto the FPGA fabric. The Y-axis on both graphs is the number of LUTs used in case of XC2VP30 device. From these graphs, we conclude that the LUT utilization increases as per the expression above.

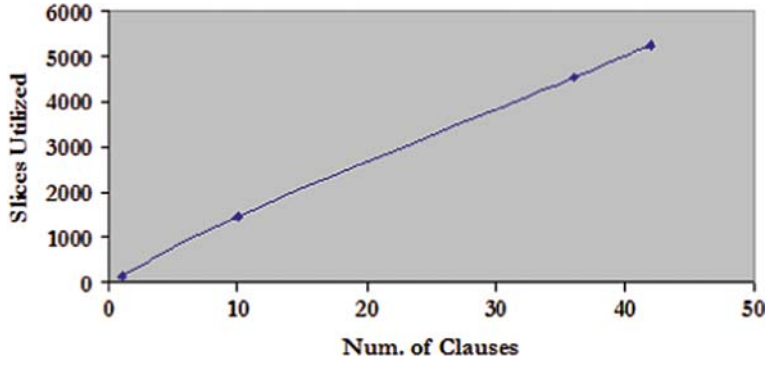


Fig. 5.3 Resource utilization for clauses

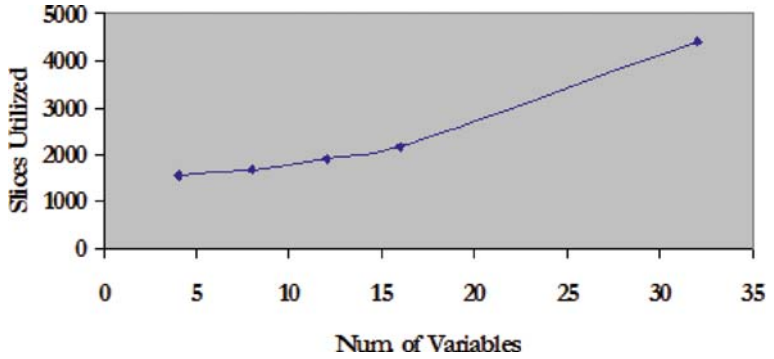


Fig. 5.4 Resource utilization for variables

The LUT utilization graphs for a Virtex-II Pro (XC2VP30) were identical to those obtained for Virtex-4 (XC4VFX140) device.

5.8.2.2 Clauses/Variable Ratio

We conducted another set of experiments to find the golden ratio (A_g), of the maximum number of clauses to the maximum number of variables in a bin. If the number of variables in a bin is too high (low) compared to the number of clauses, the *bin utilization* can be quite low. Bin utilization here is defined as follows: if a single bin is viewed as a matrix, with clauses for rows and variables for columns, bin utilization is the number of filled matrix entries over the total available matrix entries. For example, consider a bin with three clauses over three variables. If we store clauses $(a + b)$ and $(\overline{b} + c)$ in this bin, our utilization is $\frac{4}{9}$. For a set of 20 examples (taken from different CNF benchmark suites), we performed several binning runs using the cost function explained in Section 5.6. For a given number of variables we varied the number of clauses in a bin and obtained the μ, $\mu + \sigma$, and $\mu - \sigma$ of bin utilization over all the benchmarks. The number of variables was 8, 12, 16, 36, 75, and 95. Two sample plots, for number of variables equal to 16 and 36, are shown in Fig. 5.5 and 5.6, respectively. From the six plots obtained by this exercise, for a 60% bin utilization, A_g was found to be $\frac{2}{3}$.

5.8.2.3 Cycles Versus Bin Size

In order to study the effect of increasing bin size on runtime, we experimentally tried to obtain the number of hardware instructions executed as a function of bin size. We ran several satisfiable and unsatisfiable designs on our hardware platform, with different numbers of variables V and clauses $C = A_g \cdot V$ in a bin. The ratio of the total number of hardware instructions executed to the number of bins was found to be roughly constant for different $(V, A_g \cdot V)$ values. In other words, the number of hardware instructions per bin is roughly independent of the bin size. This constant was found to be $\sim$125 cycles/bin. The intuition behind this behavior is that if the bin

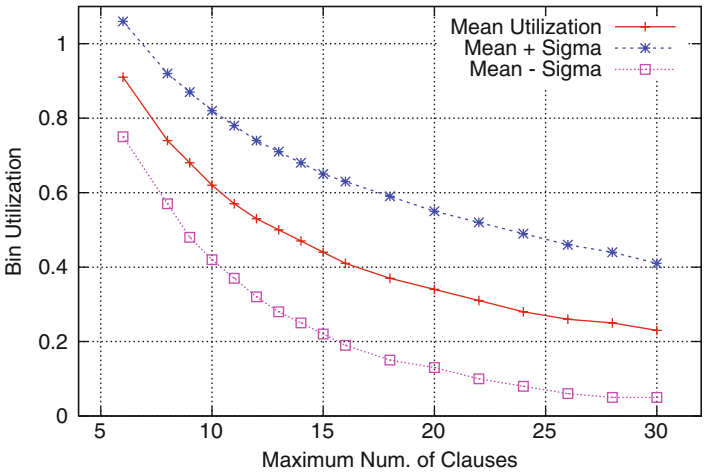


Fig. 5.5 Computing aspect ratio (16 variables)

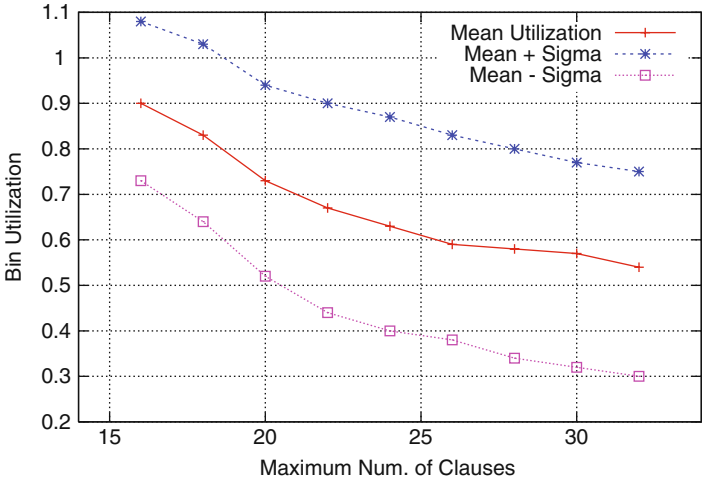


Fig. 5.6 Computing aspect ratio (36 variables)

size is large, the total number of hardware cycles decreases because the number of bins touched decreases, yielding a net constant number of hardware cycles per bin.

5.8.2.4 Bins Touched Versus Bin Size

It is important to quantify the number of bins touched as a function of the bin size. We ran several satisfiable and unsatisfiable designs, through our hardware platform, and recorded the backtracks required for completely solving the instance. For a given bin size, we simulated whether each of these backtracks would have resulted in a new bin being touched. A subset of the results is shown in Table 5.1.

Table 5.1 Number of bins touched with respect to bin size

Instance	Bins	Number of bins touched by increasing bin size			
		1×	5×	10×	20×
cmb	15	308	94	49	0
sct	57	1,193	295	174	111
cc	27	48	11	6	3
cordic	58	1,350	341	158	122
Reductions		1×	3.98×	8.33×	21.45×

Column 1 lists the instance name, while column 2 lists the number of bins obtained after bandwidth minimization. Column 3 lists the number of bins touched in order to completely solve the instance. Columns 4, 5, and 6 display the number of bins touched if the bin size is increased by 5×, 10×, and 20×, respectively. The average reduction in the number of bins touched for the last four columns is displayed in the last row. This experiment concludes that the number of bins touched reduces *linearly* with an increase in bin size.

5.8.2.5 Bin Size

Our current implementation uses a Xilinx XC2VP30 FPGA, which contains about 30K LUTs. An industry-strength implementation of our FPGA SAT solver would be done using best-in-class FPGA boards that are in the market today, which are based on the XC4VFX60 and XC4VFX140 FPGAs. These contain 60K and 140K LUTs, respectively. We therefore estimate the bin size for these boards. Table 5.2 tabulates the distribution of the LUTs in each of these devices over portions of our design that scale with bin size and also those portions of the design that do not scale with bin size. The non-scaling parts are those for which the LUT utilization does not increase while increasing the bin size. These include the Xilinx cores for DDR, DCM, PowerPC, BRAM, PLB, OPB, and the finite state machine for the decision engine. The scaling parts are those for which the device utilization increases with an increase in bin size. These include the clauses of the bin. Column 2 in Table 5.2 tabulates this distribution for our current XC2VP30-based board. Out of the total 30K available LUTS, and assuming a 70% device utilization, only 14K LUTs can be

Table 5.2 LUT distribution for FPGA devices

FPGA device	XC2VP30	XC4VFX60	XC4VFX140
Total logic cells (L)	30K	60K	140K
Xilinx cores (DDR + DCM + PowerPC + BRAM + PLB + OPB) and decision engine	7K	7K	7K
Available (0.7L–7K): clause of bin	14K	35K	91K
$(V_{\text{device}}, A_g \cdot V_{\text{device}})$	(16, 10)	(36, 24)	(75, 50)

used for storing the clauses of the bin. In case of the XC4VFX60 FPGA, as shown in column 3, about 35K LUTs can be used for the clauses of the bin. Similarly, column 4 lists the available LUTs for clauses of the bin, for the XC4VFX140 FPGA.

Using the resource utilization for a single clause and a single variable, together with the available resources for clauses of a bin, we can compute the maximum size of a bin which can be contained in the bigger FPGAs mentioned above. Say a bin of size C clauses and V variables can be configured into an FPGA device *Device*. We know that the number of LUTs of *Device* utilized for clauses of the bin is $300{\cdot}V + 20{\cdot}V \cdot C$. Since $C = \frac{2}{3} \cdot V$ based on the golden ratio A_g, we have $300{\cdot}V + 20{\cdot}\frac{2}{3} \cdot V^2$ = Available LUTs in *Device*.

Solving this quadratic equation for V gives us the size of the bin $(V, A_g \cdot V)$ that can be accommodated in any FPGA device. The last row of Table 5.2 lists the bin sizes for the FPGA devices XC2VP30, XC4VFX60, and XC4VFX140. These calculated bin sizes have been verified by synthesizing the design netlist generated for these bin sizes using the Xilinx ISE 8.2i tool, for the corresponding device.

5.8.3 Projections

Detailed runtime data (for software and hardware portions of our design) were extracted using the XC2VP30 university evaluation board. Using the performance model of the previous section, we project these runtimes for a Xilinx XC4VFX140 device.

From the performance models in Section 5.8.2, we can project the system performance (from the current implementation on the XC2VP30 device) as follows:

- Number of bins in the design are projected to grow as $\frac{V_{XC2VP30}}{V_{\text{Device}}}$.
  This is because the number of bins required for a CNF instance is inversely proportional to the bin size.
- Number of bins touched grows as $\frac{V_{XC2VP30}}{V_{\text{Device}}}$.
  From our discussion on bins touched versus bin size in Section 5.8.2, the number of bins touched is inversely proportional to bin size, which in turn is proportional to the number of variables in a bin.
- Software (PowerPC) runtimes improve as $\frac{F_{\text{Device}}}{F_{XC2VP30}} \cdot \frac{V_{\text{Device}}}{V_{XC2VP30}} \cdot 50$.
  This expression can be analyzed in three parts:
  - Software runtime is inversely proportional to the device frequency.
  - If the number of bins touched is reduced, the number of bin transfers directed by the PowerPC is reduced proportionately.
  - The bus transfer rate using Xilinx bus transfer protocols is about 50 cycles per word in our current implementation. This transfer rate can be reduced to 1 cycle per word, by writing a custom bus transfer protocol.
- Hardware (Verilog) runtimes improve as $\frac{F_{\text{Device}}}{F_{XC2VP30}} \cdot \frac{V_{\text{Device}}}{V_{XC2VP30}} \cdot \frac{(\frac{\text{cycles}}{\text{bin}})_{XC2VP30}}{(\frac{\text{cycles}}{\text{bin}})_{\text{Device}}}$.
  Again, this expression can be analyzed in three parts:

- Hardware runtime is inversely proportional to the device frequency.
- If the number of bins touched is reduced, the total number of hardware cycles required for solving the instance is reduced proportionately. This factor is $\frac{V_{\text{Device}}}{V_{XC2VP30}}$.
- The total number of hardware cycles required is proportional to the number of cycles required to solve a single bin.

Using the above expressions for the scaling of the hardware and software runtimes, the projected runtimes for a XC4VFX140-based system are shown in Table 5.3. Note that the results in Table 5.3 are obtained by taking actually the hardware and software runtimes of our XC2VP30-based platform and projecting these numbers to an industry-strength XC4VFX140-based platform. Column 1 lists the instance name, and columns 2 and 3 list the number of variables and clauses, respectively, in the instance. Column 4 lists the number of bins obtained after the CNF partitioning is performed on the host machine. Column 5 lists the number of bins 'touched' by the XC4VFX140-based hardware for solving this instance. The runtimes (in seconds) are listed in columns 6, 7, and 8. Column 6 reports the software runtime of our approach (i.e., the time taken by the PowerPC at 450 MHz to perform the bin transfers). Column 7 reports the hardware runtime (i.e., hardware runtime over all bins). The runtimes for the preprocessing step are not considered, since they are negligible with respect to the hardware or software runtime. Even if the preprocessing runtimes were higher, the time spent in partitioning the CNF instance is amply recovered when multiple SAT calls need to be made for the same instance, which commonly occurs in CAD-based SAT instances. Finally, the last column reports the MiniSAT runtimes obtained on a 3.6 GHz Pentium IV machine with 3 GB of RAM, running Linux.

Over all test cases, the net speedup over MiniSAT is 90×, and for benchmarks in which more than 4,500 bins are touched, the speedup is about 17×. Also, for benchmarks which fit in a single bin, the speedup is 2.85×10^4.

The capacity of the XC4VFX140-based system can be computed as follows. Assume that we cache 500 bins in the BRAM. Each bin has 50 variables and 75 clauses. The number of clauses C_{tot} on the number of variables V_{tot} that can be accommodated in this system is obtained by solving the following equation:

$$\text{BRAMSIZE} = (500 \cdot 50 \cdot 2 \cdot 75) + \left(\frac{C_{\text{tot}}}{A_g} \cdot \log_2(V_{\text{tot}})\right) + V_{\text{tot}} \cdot \left(4 + \log_2(V_{\text{tot}}) + \log_2\left(\frac{C_{\text{tot}}}{A_g \cdot V_{\text{Device}}}\right)\right)$$

The first term in the above equation represents the number of BRAM bits required to cache 500 bins. The second term represents the number of BRAM bits required to store the variable indices across all the bins. The third term represents the number of BRAM bits required to store the global state of all the variables in the design. This is split into three smaller terms:

- The first term requires 4 bits in total. This is to record the decision on the variable (2 bits) and the assigned/implied status of the variable (1 bit) and whether it is the earliest indexed variable that we have backtracked on (1 bit).
- The second term represents the number of bits required to record the decision level ($\log_2(V_{\text{tot}})$ bits).

Table 5.3 Runtime comparison XC4VFX140 versus MiniSAT

Instance name	Num. vars	Num. cls.	Num bins	Bins touched	Time (s)		
					PowerPC	Verilog	MiniSAT
mux_u	133	504	13	1	1.24×10^{-8}	1.43×10^{-9}	9.39×10^{-4}
cmb	62	147	4	66	6.90×10^{-6}	2.84×10^{-5}	1.01×10^{-3}
cht_u	647	2,164	48	6	9.32×10^{-7}	1.00×10^{-6}	1.97×10^{-3}
frg1_u	310	2,362	71	1	1.79×10^{-7}	5.02×10^{-7}	1.03×10^{-3}
ttt2_u	874	3,284	84	4	9.24×10^{-7}	6.77×10^{-7}	9.98×10^{-4}
term1_u	1,288	4,288	114	3	1.06×10^{-6}	3.72×10^{-7}	1.99×10^{-3}
x4_u	1,764	5,772	138	12	2.24×10^{-6}	2.67×10^{-6}	2.99×10^{-3}
x3_u	3,301	10,092	257	18	3.84×10^{-6}	3.39×10^{-6}	3.01×10^{-3}
aim-50-2_0-yes1-20	50	100	3	3	2.99×10^{-7}	1.49×10^{-6}	1.84×10^{-4}
holes6	42	133	3	4,600	5.00×10^{-4}	2.23×10^{-3}	8.98×10^{-3}
holes8	72	297	5	276,751	3.04×10^{-2}	1.21×10^{-1}	1.49
uuf100-0457	100	430	17	43,806	4.39×10^{-3}	2.58×10^{-2}	1.90×10^{-2}
uuf125-07	125	538	21	11,20,471	1.35×10^{-1}	9.03×10^{-1}	2.01×10^{-2}
Geo. mean					1.31×10^{-5}	2.27×10^{-5}	3.78×10^{-3}

- The third term represents the number of bits required to record the index of the bin in which the variable was assigned or implied, which requires as many bits as the logarithm of the number of bins ($\log_2(\frac{C_{\text{tot}}}{A_g \cdot V_{\text{Device}}})$).

The total BRAMSIZE for the XC4VFX140 part is 9.936 Mb. Solving the above equation, using a maximum number of variables (V_{tot}) of 10K, gives $C_{\text{tot}} = 280K$ clauses, the capacity of the system.

5.9 Chapter Summary

In this chapter, we have presented an FPGA-based approach for Boolean satisfiability, in which the traversal of the implication graph as well as conflict clause generation is performed in hardware, in parallel. In our approach, clauses are stored in FPGA slices. In order to solve large SAT instances, we heuristically partition the clauses into a number of bins, each of which can fit in the FPGA. This is done in a preprocessing step. The entire instance is solved using both intra- and inter-bin non-chronological backtrack, which is implemented in hardware. The on-chip BRAM is used for storing all the bins of a partitioned CNF problem. The embedded PowerPC processor on the FPGA performs the task of loading the appropriate bin from the BRAM, as requested by the hardware. Our entire flow has been verified for correctness on a Virtex-II Pro based evaluation platform. We project the runtimes obtained on this platform to an industry-strength XC4VFX140-based system and show that a speedup of 17× can be obtained over the best-in-class software approach. The projected system can handle instances with as many as 280K clauses on 10K variables.

References

1. http://www.cs.chalmers.se/cs/research/formalmethods/minisat/main.html. The MiniSAT Page
2. Abramovici, M., de Sousa, J., Saab, D.: A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In: Proceedings, Design Automation Conference (DAC), pp. 684–690 (1999)
3. Cook, S.: The complexity of theorem-proving procedures. In: Proceedings, Third ACM Symposium Theory of Computing, pp. 151–158 (1971)
4. Gulati, K., Paul, S., Khatri, S.P., Patil, S., Jas, A.: FPGA-based hardware acceleration for Boolean satisfiability. ACM Transaction on Design Automation of Electronic Systems **14**(2), 1–11 (2009)
5. Mencer, O., Platzner, M.: Dynamic circuit generation for Boolean satisfiability in an object-oriented design environment. In: HICSS '99: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 3, p. 3044. IEEE Computer Society, Washington, DC (1999)
6. Pagarani, T., Kocan, F., Saab, D., Abraham, J.: Parallel and scalable architecture for solving satisfiability on reconfigurable FPGA. In: Proceedings, IEEE Custom Integrated Circuits Conference (CICC), pp. 147–150 (2000)
7. Platzner, M., Micheli, G.D.: Acceleration of satisfiability algorithms by reconfigurable hardware. In: FPL '98: Proceedings of the 8th International Workshop on Field-Programmable

Logic and Applications, from FPGAs to Computing Paradigm, pp. 69–78. Springer-Verlag, London (1998)
8. Redekopp, M., Dandalis, A.: A parallel pipelined SAT solver for FPGAs. In: FPL '00: Proceedings of The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications, pp. 462–468. Springer-Verlag, London (2000)
9. Safar, M., El-Kharashi, M., Salem, A.: FPGA-based SAT solver. In: Proceedings, Canadian Conference on Electrical and Computer Engineering, pp. 1901–1904 (2006)
10. Safar, M., Shalan, M., El-Kharashi, M.W., Salem, A.: Interactive presentation: A shift register based clause evaluator for reconfigurable SAT solver. In: DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 153–158 (2007)
11. Silva, M., Sakallah, J.: GRASP – a new search algorithm for satisfiability. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 220–7 (1996)
12. Skliarova, I., Ferrari, A.B.: A software/reconfigurable hardware SAT solver. IEEE Transactions on Very Large Scale Integration Systems **12**(4), 408–419 (2004)
13. Suyama, T., Yokoo, M., Sawada, H., Nagoya, A.: Solving satisfiability problems using reconfigurable computing. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9**(1), 109–116 (2001)
14. Zhao, Y., Malik, S., Wang, A., Moskewicz, M., Madigan, C.: Matching architecture to application via configurable processors: A case study with Boolean satisfiability problem. In: Proceedings, International Conference on Computer Design (ICCD), pp. 447–452 (2001)
15. Zhong, P., Martonosi, M., Ashar, P.: FPGA-based SAT solver architecture with near-zero synthesis and layout overhead. IEE Proceedings – Computers and Digital Techniques **147**(3), 135–141 (2000)
16. Zhong, P., Martonosi, M., Ashar, P., Malik, S.: Accelerating Boolean Satisfiability with configurable hardware. In: Proceedings, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 186–195 (1998)

Chapter 6
Accelerating Boolean Satisfiability on a Graphics Processing Unit

6.1 Chapter Overview

In this chapter we present a Boolean satisfiability solver with a new GPU-enhanced variable ordering heuristic. Our approach is implemented in a CPU-based procedure and leverages the parallelism of a graphics processing unit (GPU). The CPU implements a complete procedure (MiniSAT), while the GPU implements an approximate procedure (an implementation of survey propagation – SurveySAT). The SAT search is initiated on the CPU, and after a user-specified fraction of decisions have been made, the GPU-based SurveySAT engine is invoked. The decisions made by this engine are returned to MiniSAT, which now updates its variable ordering by giving a higher preference to the decision variables returned by the GPU. This procedure is repeated until a solution is found. Our approach retains completeness (since it is based on a complete procedure) but has the potential of high speedup since the incomplete SurveySAT procedure that enhances the variable ordering in the complete procedure is implemented on a parallel platform. Our results demonstrate that over several satisfiable and unsatisfiable benchmarks, our technique (referred to as MESP) performs better than MiniSAT. We show a 64% speedup on average, over several benchmarks from the SAT race (2006) competition.

The rest of this chapter is organized as follows. The motivation for this work is described in Section 6.2. Section 6.3 reports some related previous work. Section 6.4 describes our SAT algorithm. This section first briefly describes our GPU-based implementation of SurveySAT. We then present the details of MESP. Experimental results are reported in Section 6.5. Section 6.6 summarizes this chapter.

6.2 Introduction

In addition to well-known complete approaches to solve SAT such as [20, 18, 13, 14] and [1], several incomplete or stochastic heuristics have been presented in the past. A partial list of these is [3, 2, 19, 7, 8]. These heuristics are iterative and usually very effective for random SAT instances. For structured SAT instances (such as those that arise out of VLSI logic circuits), their performance is mixed. For example survey

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_6,
© Springer Science+Business Media, LLC 2010

propagation based techniques [7, 8] can return a *non-convergent* or a *contradiction* result, both of which give the user no conclusive indication of the satisfiability or unsatisfiability of the instance. The advantage, however, of these incomplete techniques is that they are inherently amenable to parallelization. In this work we present a complete algorithm for Boolean satisfiability. Our algorithm implements a complete procedure (MiniSAT), which leverages the speed of an incomplete procedure (survey propagation), to augment the variable ordering heuristic of the complete procedure. Our approach retains completeness (since it implements a complete procedure) but has the potential of high speedup (since the approximate procedure is executed on a highly parallel graphics processor based platform).

This work is based on the implementation of a new variable ordering approach in a complete procedure (MiniSAT [1]), which runs on the CPU. This instance of MiniSAT is guided by a survey propagation based (SurveySAT) procedure which is implemented on the GPU. Our new algorithm is referred to as MESP (MiniSAT enhanced with survey propagation) in the sequel. The GPU is ideally suited for the (independent) variables to clauses ($V \rightarrow C$) and clauses to variables ($C \rightarrow V$) computations that need to be performed in the SurveySAT procedure. Note that in our approach, the set of clauses C on the GPU contains a subset of the recent learned clauses that were generated in MiniSAT, in addition to the original problem's clause database. Using the partial assignments of the CPU-based MiniSAT procedure, the GPU (in parallel) computes certain new variable assignments and returns these to the CPU. The CPU-based procedure now gives a higher preference to these variables during the next set of decisions it makes. The intuition behind our approach is that the assignments from the (GPU-based) survey propagation augment the variable ordering heuristic of MiniSAT with the more global view of the clause database (including recent learned clauses) that the SurveySAT procedure has. This procedure is repeated until the instance is proven satisfiable or reported as unsatisfiable. In this manner, MESP retains the best features of the 'complete' procedure and also takes advantage of a GPU-based accelerated implementation of the 'incomplete' procedure.

The key contributions of the work described in this chapter include the following:

- This is the first approach to present a CPU + GPU-based complete SAT decision procedure.
- Our SAT solver (MESP) retains the best features of a CPU-based complete SAT procedure and a GPU implementation of a highly parallel SurveySAT procedure.
- Our solver frequently refreshes the learned clause database on the GPU with the recently generated learned clauses on CPU, and thus takes advantage of the advanced learned clause generation and resolution heuristics existing in MiniSAT.
- Our GPU implementation of the SurveySAT procedure is 22× faster than a CPU-based SurveySAT implementation for several hard random benchmarks. On these random benchmarks, MiniSAT times out after several hours.

- Over several structural benchmarks from the SAT07 competition, on average MESP shows a 64% speedup when compared to MiniSAT (which was run on the CPU).

6.3 Related Previous Work

Existing SAT solvers can be categorized into *complete*, *stochastic*, *hybrid*, and *parallel* techniques. The complete techniques [10, 20, 18, 13, 14, 1] either provide *a satisfying* assignment for the SAT instance or report the instance to be *unsatisfiable*. Stochastic techniques [3, 2, 19, 7, 8] may be able to quickly provide a satisfying solution for certain SAT instances. However, they cannot prove that a SAT instance is unsatisfiable. Also, for a satisfiable instance, these solvers are not guaranteed to find a solution. Hybrid techniques [11] aim at borrowing ideas from complete and stochastic approaches to improve the overall performance. Parallel SAT solvers [6, 21, 12, 9] use multi-threaded or MIMD machines for their implementations, but require dynamic work load balancing heuristics which can be expensive. Our approach falls under the hybrid category, making use of the immense parallelism available in a GPU. Our approach is a complete technique, targeting structural SAT instances (in addition to random instances). To the best of our knowledge, there is no existing *complete* SAT solver, hybrid or otherwise, which employs the GPU for improving its performance. Some of the existing work in Boolean satisfiability is outlined next.

Among the complete approaches, the DPLL technique [10] was the first branch and search algorithm developed for solving a SAT instance. GRASP [20] augmented DPLL with *non-chronological backtracking* when a conflict was detected. SAT solvers like [18, 13, 14] inherited the features of GRASP and improved the search heuristics by employing concepts like 2-literal watching and learned clause-aging [18], improved decision strategies [13], and stronger conflict clause analysis [14]. MiniSAT [1] is a more recent SAT solver which performs a smart conflict clause simplification by applying subsumption resolution [22] and caching of intermediate results. MiniSAT has been recognized to be among the best SAT solvers in recent SAT competitions [4]. Our approach therefore employs MiniSAT as the baseline complete SAT technique and *further improves its performance* by employing a fast (albeit incomplete) SAT solver while retaining completeness.

A few examples of stochastic techniques for solving a SAT instance are discussed next. WalkSAT [3] and GSAT [2] are heuristic approaches which start by assigning a random value to each variable. If the assignment satisfies all clauses, the algorithm terminates, returning the assignment. Otherwise, a variable is flipped and the above step is repeated until all the clauses are satisfied. WalkSAT and GSAT differ in the methods used to select which variable to flip. GSAT uses a probabilistic heuristic to flip a variable, which minimizes the number of unsatisfied clauses (in the new assignment). WalkSAT first picks a clause which is unsatisfied by the current assignment, then flips a variable within that clause. This clause is generally picked at

random among unsatisfied clauses. The variable is heuristically chosen (with some probability of picking one of the variables at random), with the aim that the variable flip will result in the fewest previously satisfied clauses becoming unsatisfied. Note that WalkSAT is guaranteed to satisfy the current unsatisfied clause. WalkSAT has to do less calculation than GSAT when selecting a variable to flip, because the number of variables being considered by WalkSAT is fewer. Note that both WalkSAT and GSAT may restart with a new random assignment, if no solution has been found after several flips. This is done in order to escape out of a local minimum.

Discrete Lagrangian-based global search methods such as [19] avoid getting stuck in a local trap by using Lagrange multipliers to force the current assignment out of the current local minimum. Survey propagation [7, 8] is an iterative 'message-passing' algorithm designed to solve hard random k-SAT problems. Experimental results suggest that it may be an effective technique even for problems that are close to the hard satisfiability threshold [16]. However, it is an incomplete technique and is not effective for most hard structural SAT problems. In [17], a GPU-based implementation of survey propagation is presented. In contrast to our approach, [17] does not present a complete procedure. The authors demonstrate a $9\times$ speedup over a CPU-based implementation of survey propagation [7]. However, [17] is an incomplete procedure, frequently returning a non-convergent or contradiction result on real SAT problems which are structural. Our GPU-based implementation of survey propagation is $22\times$ faster compared to [7].

The approach of [11] is an hybrid technique which, like our approach, integrates a stochastic approach and a DPLL-based approach. A stochastic search is used to identify a subset of clauses to be passed to a DPLL SAT solver. Over several benchmarks, [11] reports on average 39% speedup against MiniSAT; however, for their unsatisfiable benchmarks their performance shows up to a $4\times$ slowdown. Our approach, on the other hand, accelerates the stochastic approach using a GPU and our results show on average 64% speedup over several satisfiable and unsatisfiable benchmarks.

Among existing parallel SAT approaches, [6] is the first parallel implementation of the DPLL procedure on a message-based MIMD machine. The input formula is dynamically divided into disjoint sub-formulas, which are solved by a DPLL-based procedure running on every processor. The approach also discusses dynamic load balancing techniques to obtain higher parallelizing efficiency. However, only random instances or unsatisfiable graph problems are discussed in the results provided by [6]. No intuition of the performance on structural SAT problems is provided. Our technique on the other hand employs a SIMD machine (GPU) for improving the performance of a complete procedure for structural *and* random SAT instances. PSATO [21] is a DPLL solver for distributed architectures, and it introduces a technique to define non-overlapping portions of the search space to be examined. Reference [15], a parallel-distributed DPLL solver, improves the workload balancing of [6] by using a master–slave communication model and work stealing. The authors emphasize the ping-pong phenomenon which may occur in workload balancing. Unlike these techniques, our technique does not require any work load balancing heuristics.

A parallel multi-threaded SAT solver is presented in [12]. It is implemented on a single multiprocessor workstation with a shared memory architecture. It shows the negative effect of parallel backtrack-search algorithm on a single multiprocessor workstation, due to increased cache misses. Our approach implements a survey propagation based technique on a SIMD GPU machine and employs it in conjunction with a complete DPLL-based solver (MiniSAT [1]). Survey propagation, as shown in the sequel, is highly amenable to parallelization and therefore allows us to obtain high overall speedups.

GridSAT [9], also a DPLL solver, is designed to run on a large number of widely distributed and heterogeneous resources: the Grid. Its key philosophy is to keep the execution as sequential as possible and to use parallelism only when required. The underlying solver is [18] and it implements a distributed learning clause database system on different but non-dedicated nationally distributed Grids. Our approach uses off-the-shelf graphics cards for accelerating Boolean satisfiability and is therefore extremely cost effective. To the best of our knowledge, there is no existing *complete* SAT solver which employs the GPU for obtaining a performance boost.

6.4 Our Approach

Our implementation of survey propagation on the GPU is explained in Section 6.4.1 and the MESP (MiniSAT enhanced with survey propagation) approach is described in Section 6.4.2.

6.4.1 SurveySAT and the GPU

In Section 6.4.1.1, we first describe the survey propagation based SAT procedure, followed by a discussion of our implementation (SurveySAT) of this approach on the GPU in Section 6.4.1.2. Finally, we present some results of our GPU-based SurveySAT engine (Section 6.4.1.3). These results are presented to illustrate the potential and the shortcomings of SurveySAT and motivate our MESP procedure.

6.4.1.1 SurveySAT

Survey propagation based SAT solvers are based on an iterative message-passing paradigm. The survey propagation algorithm is shown in Algorithm 4. Consider a SAT instance consisting of clauses C on a set of variables V. The SAT instance can be graphically represented by a *Factor Graph*, which is a bipartite graph with two kinds of nodes – *variable* nodes and *function* nodes or *clause* nodes. An undirected edge is present between variable node v and function node c iff the variable v is present in the clause c (in either polarity). The factor graph is cyclic in general, although it can be a tree. Survey propagation is exact on factor graphs that are trees [7].

Algorithm 4 Pseudocode for Survey Propagation Based SAT Solver

```
1: survey_SAT(C,V)
2: Set η's to random values
3: while converge(C,V) do
4:     Sort V in order of the absolute difference in their bias values
5:     Fix variables v* ∈ V s.t. |W_{v*}^{(+)} − W_{v*}^{(−)}| > τ. If contradiction, exit
6:     if all variables fixed then
7:         Problem SAT
8:         exit
9:     end if
10:    if Σ_{∀a,j} (η_{a→j}) < δ then
11:        call walksat()
12:    end if
13: end while
14:
15: converge(C,V)
16: repeat
17:    Compute Π's (Equations 6.1 through 6.3)
18:    Compute η's (Equation 6.4)
19:    ε = max_{∀a,j} |η_{a→j}^{old} − η_{a→j}|
20:    iter++; η^{old} ← η
21: until (iter < MAX || ε > EPS)
22: if ε ≤ EPS then
23:    return 1
24: else
25:    return 0
26: end if
```

The SurveySAT algorithm consists of clauses sending *surveys* or *messages* ($\eta_{c\to v} \in [0,1]$) to their variables. These surveys are probability values. A high value of $\eta_{c\to v}$ indicates that the clause c needs variable v to satisfy it.

For the remainder of the discussion, let i, j be variables and a, b be clauses. We denote $C(j)$ as the set of clauses that contain the variable j. Let $C_a^u(j)$ be the set of clauses that contain the variable j in the opposite polarity as it appears in clause a. Similarly, let $C_a^s(j)$ be the set of clauses that contain the variable j in the same polarity as it appears in clause a. Also, let $V(a)$ be the variables that appear in clause a.

Survey propagation begins by setting η values randomly (line 2). Then we attempt to converge on values of the variables (line 3). Each call to the convergence routine computes the survey values $\eta_{a\to i}$. To do this, we first compute three messages from each variable $j \in V(a) \setminus i$ to the clauses that contain variable j in either polarity (line 17). These message computations are shown in Equations 6.1 through 6.3. Equation (6.1) is explained below. The survey from variable j to clause a has a high value when

- other clauses (which contain variable j in the opposite polarity as clause a) have computed a high value of the survey (first square parenthesis expression) and

- other clauses (which contain variable j in the same polarity as clause a) have computed a low value of the survey (second square parenthesis expression).

Equation (6.2) can be explained similarly:

$$\Pi^{u}_{j\to a} = [1 - \prod_{b\in C^{u}_{a}(j)}(1-\eta_{b\to j})][\prod_{b\in C^{s}_{a}(j)}(1-\eta_{b\to j})] \tag{6.1}$$

$$\Pi^{s}_{j\to a} = [1 - \prod_{b\in C^{s}_{a}(j)}(1-\eta_{b\to j})][\prod_{b\in C^{u}_{a}(j)}(1-\eta_{b\to j})] \tag{6.2}$$

$$\Pi^{0}_{j\to a} = [\prod_{b\in C(j)\setminus a}(1-\eta_{b\to j})] \tag{6.3}$$

Once we have computed $\Pi^{s}_{j\to a}$, $\Pi^{u}_{j\to a}$, and $\Pi^{0}_{j\to a}$, we compute the survey $\eta_{a\to i}$ as shown in Equation (6.4) (line 18). The survey $\eta_{a\to i}$ has a large value if $\Pi^{u}_{j\to a}$ is large, thereby, clause a indicates to the variable i that it needs to be set in the polarity that would satisfy clause a. Note that if $V(a) \setminus i$ is empty, then $\eta_{a\to i} = 1$:

$$\eta_{a\to i} = \prod_{j\in V(a)\setminus i}\left[\frac{\Pi^{u}_{j\to a}}{\Pi^{u}_{j\to a} + \Pi^{s}_{j\to a} + \Pi^{0}_{j\to a}}\right] \tag{6.4}$$

Note that if any of the sets $C^{s}_{a}(j)$, $C^{u}_{a}(j)$, or $C(j)$ are empty, then their corresponding product term takes on a value 1. Equation (6.3) in the denominator of Equation (6.4) avoids a possibility of a division by 0 in Equation (6.4).

After computing the ηs, we check for convergence, by computing the maximum of the absolute value of the difference between $\eta_{a\to i}$ and $\eta^{\text{old}}_{a\to i}$ (from the last iteration) in the *converge()* routine (line 19). If the largest entry of this vector of absolute differences is smaller than a user-defined value EPS (line 22), then we declare convergence (line 23). If convergence has not occurred after MAX iterations, we return a 0 (line 25) and the *survey_SAT()* returns unsuccessfully (line 3) with a *nonconvergent* status. The *converge()* routine is iterated until convergence is achieved or a user-specified number of iterations MAX is reached. We use MAX = 1,000 in all our experiments and $EPS = 0.01$.

Upon convergence, we compute two bias values for each variable and sort the variable list in the descending order of the absolute difference in their bias values (line 4). There are two biases $W^{(+)}_{i}$ and $W^{(-)}_{i}$ that are computed, as shown in Equations (6.5) and (6.6). The intuition behind the computation of these two values is similar to that of the computation of surveys $\eta_{a\to i}$, except for the fact that the biases are computed for each variable. Also, the Π values that the bias computations are based on (Equations (6.7) through (6.9)) are computed for all clauses $C_{+}(i)$ ($C_{-}(i)$) which contain the variable i in the positive (negative) polarity, using the converged

values of the surveys ($\eta^*_{a\to i}$). $C(i)$ is the set of clauses which contain the variable i in either polarity.

$$W_i^{(+)} = \frac{\hat{\Pi}_i^+}{\hat{\Pi}_i^+ + \hat{\Pi}_i^- + \hat{\Pi}_i^0} \tag{6.5}$$

$$W_i^{(-)} = \frac{\hat{\Pi}_i^-}{\hat{\Pi}_i^+ + \hat{\Pi}_i^- + \hat{\Pi}_i^0} \tag{6.6}$$

$$\hat{\Pi}_i^+ = [1 - \prod_{a\in C_+(i)}(1 - \eta^*_{a\to i})][\prod_{a\in C_-(i)}(1 - \eta^*_{a\to i})] \tag{6.7}$$

$$\hat{\Pi}_i^- = [1 - \prod_{a\in C_-(i)}(1 - \eta^*_{a\to i})][\prod_{a\in C_+(i)}(1 - \eta^*_{a\to i})] \tag{6.8}$$

$$\hat{\Pi}_i^0 = [\prod_{a\in C(i)}(1 - \eta^*_{a\to i})] \tag{6.9}$$

All variables with the absolute difference in bias values $|W_i^{(+)} - W_i^{(-)}| > \tau$ (a user-specified value) are *fixed* (line 5). If all variables are fixed, then the problem is SAT and declared as such and we exit (lines 6–8). If all surveys are trivial (line 10) then we call a local search process (*WalkSAT()* [3] in this instance). If neither condition above holds, we run the *converge()* routine again. In subsequent runs of the converge routine, variables that were previously fixed do not participate in the computation of Πs (Equations (6.1) through (6.3) and (6.7) through (6.9)). Similarly, clauses that are satisfied as a consequence of fixing some variable do not participate in the computation of $\eta_{a\to i}$ and the bias values (Equations (6.4), (6.5), and (6.6)).

Note that the *survey_SAT* algorithm can fail in two ways – it can fail to achieve convergence or it can converge such that the set of fixed variables is inconsistent although the problem is satisfiable (returning a *contradiction* status in this case).

6.4.1.2 SurveySAT on the GPU

Note that the *survey_SAT()* procedure is naturally amenable for GPU implementation. Both the Π and η computations are inherently parallelizable since the Π and η values are computed using independent data. In our implementation of *survey_SAT()* on the GPU, we restrict the SAT instance to be a 3-SAT instance. We compute Πs (line 17) by issuing $|V|$ parallel threads on the GPU, followed by a

thread synchronization command. Next we compute the surveys $\eta_{a\to i}$ (line 18) by issuing $|C|$ threads on the GPU (each of which computes the $\eta_{a\to i}$ values for all the three variables in its clause). The convergence check (line 19) is performed by computing a sum $Z = \sum_{\forall a,j} [(|\eta^{old}_{a\to j} - \eta_{a\to j}|) \leq EPS?0:1]$. If any $\eta_{a\to j}$ has not converged, then $Z > 0$. Hence convergence is checked by computing Z using an integer add operation over all variables in all the clauses, using a reduction-based addition subroutine. On the GPU, line 21 similarly becomes

until (iter $<$ MAX $||$ $Z > 0$)

Also, the check of line 22 becomes

if $Z = 0$ **then**.

The test for trivial convergence (when all ηs are close to 0) (line 10) is performed using a reduction-based floating point add operation on the GPU. Both bias values (for all variables) are computed by issuing $2|V|$ threads on the GPU, and they are sorted using a parallel bitonic sorting operation on the GPU [5] (line 4). The fixing of variables (line 5) is performed on the CPU.

The data structures on the GPU corresponding to the SAT instance are shown in Fig. 6.1. The static information about the SAT instance is stored in two sets of arrays:

- Static per-variable data is stored in three arrays. Each array is indexed by the variable number. For each variable, the arrays store the indices of the clauses it appears in, the polarity of each appearance, and the literal number of this variable in each clause that it appears in.
- Static per-clause data is stored in two arrays. Note that each clause has at most three variables. Each array is indexed by the clause number. For each clause, the two arrays store the variable index and polarity of each literal in that clause.

There are two additional sets of arrays that store the information computed during the *survey_SAT* computations. The first set of two arrays stores the $\prod_{b\in C_+(j)} (1-\eta_{b\to j})$ and $\prod_{b\in C_-(j)} (1-\eta_{b\to j})$ values for each variable. These arrays are written by the variables and read by the clauses. Another array stores the $\eta_{a\to j}$ values, which are written by the clauses and read by the variables.

All the above data is stored in global memory on the GPU. Note that there is a single burst transfer from the CPU to the GPU, to transfer the static information mentioned above. During the computation of the Π and η quantities, there are no transfers between the GPU and the CPU. The information that is transferred from the GPU to the CPU is the list of variables sorted in decreasing order of the absolute difference of their bias values ($|W_i^{(+)} - W_i^{(-)}|$). After the CPU has fixed any variables, it returns to the GPU a list of variables that are fixed (these do not participate in the η computations any more) and the clauses that are satisfied as a result (these do not participate in Π computations any more).

The size of thread blocks on the GPU must be a multiple of 32. As a result, all the arrays shown in Fig. 6.1 are padded to the next highest multiple of 32 in our implementation. The reduction-based add and sort operations are most efficient

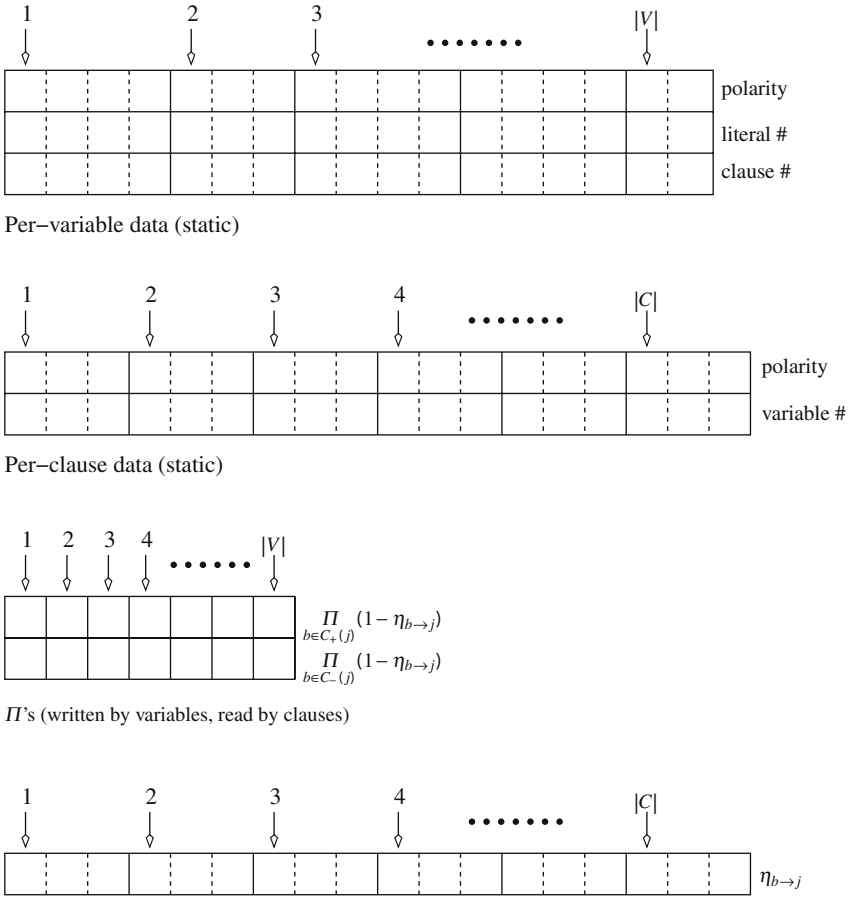


Fig. 6.1 Data structure of the SAT instance on the GPU

for arrays whose size is a power of 2. For this reason, the arrays of the absolute difference of bias values and the predicate value of $|\eta^{\text{old}}_{a \to j} - \eta_{a \to j}| \leq EPS$ are also padded to the next highest power of 2.

The NVIDIA GTX 280 has 1 GB of on-board memory. With the above memory organization, we can easily fit SAT instances with up to 1M variables and 10M clauses.

6.4.1.3 SurveySAT Results on the GPU

The SurveySAT algorithm described in Section 6.4.1.2 was implemented in CUDA. It was run on a GTX 280 GPU card from NVIDIA which has 1 GB on-board (global) memory and runs at a frequency of 1.4 GHz. The results obtained for SurveySAT

(on the GPU) were compared against a CPU implementation of SurveySAT [7] and MiniSAT which was also run on the CPU. The CPU used in our experiments is a 2.67 GHz, Intel *i*7 processor with 9 GB RAM, and running Linux.

Table 6.1 compares MiniSAT (on the CPU) with SurveySAT (on the CPU) and SurveySAT (on the GPU) over five random and three structural benchmarks. Column 1 lists the random and structural benchmarks. All random benchmarks are satisfiable. The first structural problem is satisfiable and the remaining two are unsatisfiable. Columns 2 and 3 report the number of variables and clauses in each of the benchmarks. Column 4 reports the MiniSAT runtimes (in seconds) on these benchmarks on the CPU and GPU, respectively. Columns 5 and 6 report the SurveySAT runtimes (in seconds) for the same benchmarks on the CPU and GPU, respectively. A '–' implies that the procedure either did not converge in MAX iterations (MAX = 1,000) or reported a contradiction. Column 6 reports the speedup of SurveySAT on the GPU compared to SurveySAT on the CPU.

For random benchmarks, SurveySAT is several orders of magnitude faster than MiniSAT; however, for structural examples the performance is mixed. In particular, for unsatisfiable benchmarks, the response from SurveySAT (on the CPU or the GPU) is non-conclusive. Our GPU-based SurveySAT is on average 22× faster than the CPU implementation of SurveySAT, over the instances for which SurveySAT successfully completes. In summary, even though SurveySAT can perform extremely well for random instances, for structural instances its performance is mixed, and therefore the technique is not useful for practical SAT instances. In the next section, we discuss our algorithm that retains the completeness of MiniSAT, while speeding it up with guidance obtained by SurveySAT (on the GPU).

6.4.2 MiniSAT Enhanced with Survey Propagation (MESP)

In our MESP approach we implement a CPU-based complete SAT solver with a new GPU-enhanced variable ordering heuristic. In MiniSAT, the inbuilt variable ordering heuristic (which determines what variable will be assigned next) is the variable state independent decaying sum (VSIDS) heuristic. VSIDS makes a decision based on the *activity* value of a variable. The activity is a literal occurrence count, with a higher weight placed on variables of the more recently added clauses. The activity of all variables present in the resolvent clauses, during conflict resolution and learned clause generation, is incremented by fixed amount F_m. If any variable's score becomes too high, the activity of all variables is uniformly decayed. In MESP, we update the activities of certain variables based on the guidance obtained from the (incomplete) survey propagation (on the GPU). This is explained next.

In MESP, we first start the search in MiniSAT, after reading in the given SAT instance. The SAT instance is also copied over to the GPU, organized in the manner illustrated in Fig. 6.1. After MiniSAT has made some progress (measured by whether the number of decisions it has made equals $D\%$ of the number of variables in the instance), it makes a call to SurveySAT. MiniSAT transfers the current

Table 6.1 Comparing MiniSAT with SurveySAT (CPU) and SurveySAT (GPU)

Benchmark	Num. var	Num. cl	MiniSAT (s)	SurveySAT (CPU) (s)	SurveySAT (GPU) (s)	Speedup
Random_1	20,000	83,999	> 2 h	3,009.67	172.87	17.41×
Random_2	16,000	67,199	> 2 h	1,729.48	110.60	15.63×
Random_3	12,000	50,399	> 2 h	1,002.48	57.98	17.29×
Random_4	8,000	33,599	> 2 h	369.61	5.82	63.80×
Random_5	4,000	16,799	> 2 h	65.01	3.69	17.617
uf200-097	200	860	0.15	0.20	0.08	2.50
hole10	187	792	1.3	–	–	
uuf200-018	200	860	0.15	–	–	
Average						22.37×

assignments and a subset of the recent learned clauses onto the GPU. In our implementation, learned clauses with length less than 50 literals are transferred to the GPU. We augment the clause database on the GPU with 3 sets of learned clauses (set C_1 with $\geq$0 and $<$10 literals, set C_2 with $\geq$10 and $<$25 literals, and set C_3 with $\geq$25 and $<$50 literals). Storage for learned clauses is statically allocated in the global memory on the GPU. The routine *converge(C,V)* in SurveySAT is now modified to *converge(C, C_1, C_2, C_3, V)*, where the η computations (over the clauses) are done in four separate kernels. Note that the η computation over all clauses is not done as a single kernel in order to avoid underutilized threads due to the large variance in the length of the learned clauses. Further, unless at least 256 learned clauses are transferred to the GPU in any of the 3 sets, the kernel for η computation for the corresponding set is not invoked. The number of clauses in each set C_i was set to 8K.

After SurveySAT has converged and fixed a set of variables U (variables whose absolute difference of bias values is greater than τ) on the GPU, it returns. MiniSAT now increments the activity of all variables in the set U by F_{sp} and continues with its search. The idea is that since the instance converged (over all clauses as well as a subset of the recent learned clauses) in SurveySAT by fixing the variables in set U with no contradiction, an earlier decision on the variables in U would enable a better search in the CPU-based MiniSAT procedure.

After MiniSAT makes more decisions (and implications), and another D% of the number of variables in the instance have been decided, the GPU-based survey propagation algorithm is invoked again. The total number of such *calls* to the SurveySAT routine is limited to P, which is user specified. At every invocation of SurveySAT, any existing variable assignments on the GPU are erased.

In the original MiniSAT approach, after a fixed number of conflicts R are detected (or learned clauses are computed), the solver is restarted from the root of the decision tree. This is done in order to allow the solver to start afresh with the guidance of the learned clauses and the activities of the variables. Also the number of allowed conflicts R is incremented by a factor of 1.5 upon each such restart. In our approach, each time SurveySAT is invoked, the current allowable number of conflicts (or the maximum number of stored learned clauses) in the MiniSAT portion of MESP is incremented (by a factor of 1.5, based on the existing strategy in MiniSAT). Thus our solver is not 'restarted' from the root of the decision tree as often as the CPU-based MiniSAT.

When the SurveySAT routine returns from the GPU after the ith call, four outcomes are possible:

- The SurveySAT routine converges, and based on the absolute difference of the biases of each variable, a set of variables U is found. These variables are passed along to MiniSAT, which now increments their activity by an amount F_{sp} and continues the search.
- The SurveySAT routine converges, and based on the absolute difference of the biases of each variable, no variables can be fixed. In this case, it returns an empty

set U to MiniSAT. In this case MiniSAT continues its search as it would have if there had been no call to SurveySAT.
- The SurveySAT routine does not converge, or converges to a state which is inconsistent. In this case also it returns an empty set U to MiniSAT.
- The SurveySAT routine converges and heuristically determines that the factor graph is a tree. On calling *WalkSAT*, if a satisfying solution is found we are done and the satisfiability of the instance is determined by SurveySAT. If *WalkSAT* is unable to find a satisfying solution, the SurveySAT routine returns the set U to MiniSAT. The CPU-based MiniSAT now increments the activity of the variables in the set U by F_{sp} and continues its search.

In the next section we discuss the experimental setup and compare the performance of MESP to MiniSAT.

6.5 Experimental Results

Table 6.2 compares the performance of our MESP technique with MiniSAT [1] on several structural instances (both satisfiable and unsatisfiable) from the SAT RACE 2006 and SAT 2004 [4] benchmark suite. The CPU used in our experiments is a 2.67 GHz, Intel *i*7 processor with 9 GB RAM, running Linux. The GPU used is the NVIDIA GeForce 280 GTX.

Columns 1 lists the benchmark name and column 2 reports if the instance is satisfiable or unsatisfiable. The numbers of variables and clauses in the original instance (referred to as k-SAT) are reported in columns 3 and 4, respectively. Column 5 reports the MiniSAT runtime on the k-SAT version of the example (in seconds). All k-SAT instances are converted to 3-SAT using a Perl script, before we can run MESP. This is because MESP only handles 3-SAT instances. The numbers of variables and clauses in the 3-SAT version of the instances are reported in columns 6 and 7. The MiniSAT runtime (in seconds) for the 3-SAT version of the problem is reported in column 8. Column 9 reports the runtime (in seconds) of the MESP approach, on the 3-SAT version of the problem. Columns 10 and 11 report the ratio of the runtimes of MiniSAT (on the k-SAT instance) to MESP and of MiniSAT (on the 3-SAT instance) to MESP, respectively.

The various parameters of MESP were set as follows: MAX = 1,000, EPS = 0.01, $\tau = 0.1$, $D = 1$, $F_{sp} = F_m = 1$, and maximum number of GPU calls (P) = 20. In MESP we refreshed the learned clauses on the GPU on every 5th invocation of SurveySAT. During the other invocations the learned clauses from a previous iteration were used. On the GPU we statically allocate memory for 3 sets of 8K learned clauses, of length <10 literals, ≥ 10 literals and <25 literals, and ≥ 25 literals and <50 literals. In all our benchmarks, the final decision of reporting the instance to be satisfiable or unsatisfiable was made by the MiniSAT (CPU) portion of MESP. In other words, for these structural benchmarks, the SurveySAT routine was never able to exit early by determining a satisfying assignment using *WalkSAT*.

Table 6.2 Comparing MESP with MiniSAT

		k-SAT			3-SAT				Speedup over	
Benchmark	S/U	# Vars.	# Cls.	MiniSAT (k) (sec)	# Vars.	# Cls.	MiniSAT (3) (sec)	MESP (sec)	MiniSAT (k)	MiniSAT (3)
139464p22	S	327,932	1,283,772	29.84	530,027	1,890,057	39.58	15.28	1.95×	2.59×
AProVE07-04	U	78,607	208,911	110.39	104,732	287,286	166.25	95.91	1.15×	1.73×
AProVE07-15	U	45,672	97,451	46.20	50,711	112,568	92.06	113.16	0.41×	0.81×
eijk.bs4863.S.aig-20	S	74,044	276,119	12.58	118,092	408,263	14.47	16.77	0.75×	0.86×
eijk.bs4863.S.aig-30	S	140,089	530,249	487.98	234,412	813,218	619.03	181.86	2.68×	3.40×
eijk.S298.S	U	73,222	283,211	8.42	136,731	473,738	10.01	8.47	0.99×	1.18×
Intel-034.aig.smv-10	U	173,475	593,345	18.39	274,460	896,300	27.83	32.15	0.57×	0.87×
spec10-and-env-10	U	100,444	593,345	17.07	105,949	668,100	23.00	30.81	0.55×	0.75×
t22-034-10.aig.cnf	U	12,714	50,237	24.96	13,401	52,789	27.95	8.20	3.04×	3.41×
vis.arbiter.E-50	U	12,683	48,336	24.87	13,191	49,860	26.05	7.66	3.25×	3.40×
hole10.cnf	U	187	792	1.30	187	792	1.30	1.03	1.26×	1.26 ×
par16-3.cnf	S	1,015	3,344	0.15	1,015	3,344	0.15	0.09	1.67×	1.67×
uf200-097.cnf	S	200	860	0.15	200	860	0.15	0.05	3.00×	3.00×
Average									1.64×	1.92×

Over our benchmarks, on average, MESP for the 3-SAT version of the instances showed a 64% speedup compared to MiniSAT which was run on the original problem instances (k-SAT). When compared to MiniSAT runtimes for the 3-SAT version of the SAT instances, MESP is on average about $2\times$ faster. We could have implemented our SurveySAT approach on the GPU with the maximum length of the (regular) clauses being >3 and obtained higher speedups in comparison to the MiniSAT runtimes for the original (k-SAT) version of the instances.

6.6 Chapter Summary

In this chapter, we have presented a complete Boolean satisfiability approach with a new GPU-enhanced variable ordering heuristic. Our approach is implemented in a CPU-based complete procedure, which leverages the parallelism of a GPU to aid the complete algorithm. The CPU implements MiniSAT, a complete procedure, while the GPU implements SurveySAT, an approximate procedure. When a problem instance is read in, the SAT search is initiated on the CPU. After a user-specified fraction of decisions have been made, the GPU-based SurveySAT engine is invoked. The decisions, if any, made by this engine are returned to MiniSAT, which now updates its variable ordering by incrementing the activity of the decision variables returned by the GPU. This procedure is repeated until a solution is found. Our approach retains completeness (since it implements a complete procedure) but has the potential of high speedup (since the incomplete procedure is executed on a highly parallel graphics processor platform). Experimental results demonstrate that over several satisfiable and unsatisfiable benchmarks, our approach performs better than MiniSAT. On average, we demonstrate a 64% speedup over several benchmarks when compared with MiniSAT runtimes (MiniSAT was run on the original versions of the instances). When compared with MiniSAT runtimes for the 3-SAT version of the problems, our approach yields a speedup of about $2\times$.

References

1. http://www.cs.chalmers.se/cs/research/formalmethods/minisat/main.html. The MiniSAT Page
2. http://www.cs.rochester.edu/u/kautz/papers/gsat. GSAT-USERS-GUIDE
3. http://www.cs.rochester.edu/u/kautz/walksat. WalkSAT homepage
4. http://www.satcompetition.org/. The International SAT Competitions Web Page
5. NVIDIA CUDA Homepage. `http://developer.nvidia.com/object/cuda.html`
6. Bohm, M., Speckenmeyer, E.: A fast parallel SAT-solver – Efficient workload balancing. In: Annals of Mathematics and Artificial Intelligence, pp. 40–0 (1996)
7. Braunstein, A., Mezard, M., Zecchin, R.: Survey propagation: An algorithm for satisfiability. Random Structures and Algorithms **27**, 201–226 (2005)
8. Chavas, J., Furtlehner, C., Mezard, M., Zecchina, R.: Survey-propagation decimation through distributed local computations. Journal of Statistical Mechanics (11), 11016 (2005)
9. Chrabakh, W., Wolski, R.: GridSAT: A Chaff-based distributed SAT solver for the grid. In: SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, p. 37. IEEE Computer Society, Washington, DC (2003)

10. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communication of the ACM **5**(7), 394–397 (1962)
11. Fang, L., Hsiao, M.S.: A new hybrid solution to boost SAT solver performance. In: DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1307–1313. EDA Consortium, San Jose, CA (2007)
12. Feldman, Y.: Parallel multithreaded satisfiability solver: Design and implementation (2005)
13. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Proceedings, Design, Automation and Test in Europe (DATE) Conference, pp. 142–149 (2002)
14. Jin, H., Awedh, M., Somenzi, F.: CirCUs: A satisfiability solver geared towards bounded model checking. In: Computer Aided Verification, pp. 519–522 (2004)
15. Jurkowiak, B., Li, C.M., Utard, G.: A parallelization scheme based on work stealing for a class of SAT solvers. J. Autom. Reason. **34**(1), 73–101 (2005)
16. Maneva, E., Mossel, E., Wainwright, M.J.: A new look at survey propagation and its generalizations. In: SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1089–1098. Society for Industrial and Applied Mathematics, Philadelphia, PA (2005)
17. Manolis, P., Zhang, Y.: Implementing survey propagation on graphics processing units. In: SAT '06: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, pp. 311–324 (2006)
18. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the Design Automation Conference, pp. 530–535 (2001)
19. Shang, Y.: A discrete Lagrangian-based global-search method for solving satisfiability problems. Journal of Global Optimization **12**, 61–99 (1998)
20. Silva, M., Sakallah, J.: GRASP-a new search algorithm for satisfiability. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 220–7 (1996)
21. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21**(4-6), 543–560 (1996)
22. Zheng, L., Stuckey, P.J.: Improving SAT using 2SAT. In: ACSC '02: Proceedings of the Twenty-fifth Australasian Conference on Computer Science, pp. 331–340. Australian Computer Society, Inc., Darlinghurst, Australia (2002)

Part III
Control Plus Data Parallel Applications

Outline of Part III

In Part I of this monograph, we discussed candidate hardware platforms for EDA algorithm acceleration. In Part II, we presented approaches to accelerate Boolean satisfiability (SAT), a control-dominated EDA application. We used three hardware platforms – a custom IC, an FPGA, and a GPU – for accelerating SAT. In Part III of this monograph, we present the acceleration of several EDA applications, with varying degrees of inherent parallelisms. In particular, we accelerated the following applications using GPUs:

- Statistical Static Timing Analysis

  With the diminishing minimum feature sizes of VLSI fabrication processes, the impact of process variations is becoming increasingly significant. The resulting increase in delay variations significantly affects the timing yield and the maximum operating frequency of designs. Static timing analysis (STA) is heavily used in a conventional VLSI design flow to estimate circuit delay and the maximum operating frequency of the design. Statistical STA (SSTA) was developed to include the effect of process variations, in order to analyze circuit delay more accurately. Monte Carlo based SSTA is a simple and accurate method of performing SSTA. However, its main drawback is its high runtime. We exploit the inherent parallelism in Monte Carlo based SSTA and present its implementation on a GPU in Chapter 7. In this approach we map Monte Carlo based SSTA to the large number of threads that can be computed in parallel on a GPU. Our approach performs multiple delay simulations of a single gate in parallel. Our approach further benefits from a parallel implementation of the Mersenne Twister pseudo-random number generator on the GPU, followed by Box – Muller transformations (also implemented on the GPU). These are used for generating gate delay numbers from a normal distribution. We only need to store the μ and σ of the pin-to-output delay distributions for all inputs and for every gate. This data is stored in fast cached memory on the GPU, and we thereby leverage the large memory bandwidth of the GPU. All threads compute identical instructions, but on different data, with no control or data dependency, as required by the SIMD programming semantics of the GPU. Our approach is implemented on an

NVIDIA GeForce GTX 280 GPU card. Experimental results indicate that this approach can obtain an average speedup of about 818× as compared to a serial CPU implementation. With the recently announced cards with quad GTX 280 GPUs, we estimate that our approach would attain a speedup of over 2,400×.

- Accelerating Fault Simulation on a Graphics Processor

  In today's complex digital designs, with possibly several million gates, the number of faulty variations of the design can be dramatically higher. Fault simulation is an important but expensive step of the VLSI design flow, and it helps to identify faulty designs. Given a digital design and a set of input vectors V defined over its primary inputs, fault simulation evaluates the number of stuck-at faults F_{sim} that are tested by applying the vectors V. The ratio of F_{sim} to the total number of faults in the design F_{total} is a measure of the fault coverage. The task of finding this ratio is often referred to as *fault grading* in the industry. Given the high computational cost for fault simulation, it is extremely important to explore ways to accelerate this application. The ideal fault simulation approach should be fast, scalable, *and* cost effective. In Chapter 8, we study the acceleration of fault simulation on a GPU. Fault simulation is inherently parallelizable, and the large number of threads that can be executed in parallel on a GPU can be employed to perform a large number of gate evaluations in parallel. We implement a pattern and fault parallel fault simulator, which fault-simulates a circuit in a levelized fashion. We ensure that all threads of the GPU compute identical instructions, but on different data. Fault injection is also performed along with gate evaluation, with each thread using a different fault injection mask. Since GPUs have an extremely large memory bandwidth, we implement each of our fault simulation threads (which execute in parallel with no data dependencies) using memory lookup. Our experiments indicate that our approach, implemented on a single NVIDIA GeForce GTX 280 GPU card, can simulate on average 47× faster when compared to an industrial fault simulator. On a Tesla (8-GPU) system, our approach is potentially 300× faster.
- Fault Table Generation Using a Graphics Processor

  A fault table is essential for fault diagnosis during VLSI testing and debug. Generating a fault table requires extensive fault simulation, with no fault dropping. This is extremely expensive from a computational standpoint. We explore the generation of a fault table using a GPU in Chapter 9. We employ a pattern parallel approach, which utilizes both bit parallelism and thread-level parallelism. Our implementation is a significantly modified version of FSIM, which is pattern parallel fault simulation approach for single-core processors. Like FSIM, our approach utilizes critical path tracing and the dominator concept to reduce runtime by pruning unnecessary simulations. Further modifications to FSIM allow us to maximally harness the GPU's immense memory bandwidth and high computational power. In this approach we do not store the circuit (or any part of the circuit) on the GPU. We implement efficient parallel reduction operations to speed up fault table generation. In comparison to FSIM∗, which is FSIM modified to generate a fault table on a single-core processor, our approach on a single NVIDIA Quadro FX 5800 GPU card can generate a fault table 15× faster on

average. On a Tesla (8-GPU) system, our approach can potentially generate the same fault table 90× faster.

- Fast Circuit Simulation Using Graphics Processor

  SPICE-based circuit simulation is a traditional workhorse in the VLSI design process. Given the pivotal role of SPICE in the IC design flow, there has been significant interest in accelerating SPICE. Since a large fraction (on average 75%) of the SPICE runtime is spent in evaluating transistor model equations, a significant speedup can be availed if these evaluations are accelerated. We study the speedup obtained by implementing the transistor model evaluation on a GPU and porting it to a commercial fast SPICE tool in Chapter 10. Our experiments demonstrate that significant speedups (2.36× on average) can be obtained for the commercial fast SPICE tool. The asymptotic speedup that can be obtained is about 4×. We demonstrate that with circuits consisting of as few as 1,000 transistors, speedups in the neighborhood of this asymptotic value can be obtained.

Chapter 7
Accelerating Statistical Static Timing Analysis Using Graphics Processors

7.1 Chapter Overview

In this chapter, we explore the implementation of Monte Carlo based statistical static timing analysis (SSTA) on a graphics processing unit (GPU). SSTA via Monte Carlo simulations is a computationally expensive, but important step required to achieve design timing closure. It provides an accurate estimate of delay variations and their impact on design yield. The large number of threads that can be computed in parallel on a GPU suggests a natural fit for the problem of Monte Carlo based SSTA to the GPU platform. Our implementation performs multiple delay simulations for a single gate in parallel. A parallel implementation of the Mersenne Twister pseudo-random number generator on the GPU, followed by Box–Muller transformations (also implemented on the GPU), is used for generating gate delay numbers from a normal distribution. The μ and σ of the pin-to-output delay distributions for all inputs of every gate are obtained using a memory lookup, which benefits from the large memory bandwidth of the GPU. Threads which execute in parallel have no data/control dependencies on each other. All threads compute identical instructions, but on different data, as required by the single instruction multiple data (SIMD) programming semantics of the GPU. Our approach is implemented on an NVIDIA GeForce GTX 280 GPU card. Our results indicate that our approach can obtain an average speedup of about 818× as compared to a serial CPU implementation. With the quad GTX 280 GPU [6] cards, we estimate that our approach would attain a speedup of over 2,400×. The correctness of the Monte Carlo based SSTA implemented on a GPU has been verified by comparing its results with a CPU-based implementation.

The remainder of this chapter is organized as follows. Section 7.2 discusses the motivation behind this work. Some previous work in SSTA has been described in Section 7.3. Section 7.4 details our approach for implementing Monte Carlo based SSTA on GPUs. In Section 7.5 we present results from experiments which were conducted in order to benchmark our approach. We summarize this chapter in Section 7.6.

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_7,
© Springer Science+Business Media, LLC 2010

7.2 Introduction

The impact of process variations on the timing characteristics of VLSI design is becoming increasingly significant as the minimum feature sizes of VLSI fabrication processes decrease. In particular, the resulting increase of delay variations strongly affects timing yield and reduces the maximum operating frequency of designs. Processing variations can be random or systematic. Random variations are independent of the locations of transistors within a chip. An example is the variation of dopant impurity densities in the transistor diffusion regions. Systematic variations are dependent on locations, for example exposure pattern variations and silicon-surface flatness variations.

Static timing analysis (STA) is used in a conventional VLSI design flow to estimate circuit delay, from which the maximum operating frequency of the design is estimated. In order to deal with variations and overcome the limitations due to the deterministic nature of traditional STA techniques, *statistical* STA (SSTA) was developed. The main goal of SSTA is to include the effect of process variations and analyze circuit delay more accurately. Monte Carlo based SSTA is a simple and accurate method for performing SSTA. This method generates N samples of the gate delay random variable (for each gate) and executes static timing analysis runs for the circuit using each of the N sets of the gate delay samples. Finally, the results are aggregated to produce the delay distribution for the entire circuit. Such a method is compatible with the process variation data obtained from the fab line, which is essentially in the form of samples of the process random variables. Another attractive property of Monte Carlo based SSTA is the high level of accuracy of the results. However, its main drawback is the high runtime. We demonstrate that Monte Carlo based SSTA can be effectively implemented on a GPU. We obtain a $818\times$ *speedup* in the runtime, with no loss of accuracy. Our speedup numbers include the time incurred in transferring data to and from the GPU.

Any application which has several independent computations that can be issued in parallel is a natural match for the GPU's SIMD operational semantics. Monte Carlo based SSTA fits this requirement well, since the generation of samples and the static timing analysis computations for a single gate can be executed in parallel, with no data dependency. We refer to this as *sample parallelism*. Further, gates at the same logic level can execute Monte Carlo based SSTA in parallel, without any data dependencies. We call this *data parallelism*. Employing sample parallelism and data parallelism simultaneously allows us to maximally exploit the high memory bandwidths of the GPU, as well as the presence of hundreds of processing elements on the GPU. In order to generate the random samples, the *Mersenne Twister* [22] pseudo-random number generator is employed. This pseudo-random number generator can be implemented in a SIMD fashion on the GPU, and thus is well suited for our Monte Carlo based SSTA engine. The μ and σ for the pin-to-output falling (and rising) delay distributions are stored in a lookup table (LUT) in the GPU device memory, for every input of every gate. The large memory bandwidth allows us to perform lookups extremely fast. The SIMD computing paradigm of the GPU is thus maximally exploited in our Monte Carlo based SSTA implementation.

In this work we have only considered uncorrelated random variables while implementing SSTA. Our current approach can be easily extended to incorporate spatial correlations between the random variables, by using principal component analysis (PCA) to transform the original space into a space of uncorrelated principal components. PCA is heavily used in multivariate statistics. In this technique, the rotation of axes of a multidimensional space is performed such that the variations, projected on the new set of axes, behave in an uncorrelated fashion. The computational techniques for performing PCA have been implemented in a parallel (SIMD) paradigm, as shown in [18, 13].

Although our current implementation does not incorporate the effect of input slew and output loading effects while computing the delay and slew at the output of a gate, these effects can be easily incorporated. Instead of storing just a pair of (μ and σ) values for each pin-to-output delay distribution for every input of every gate, we can store $K \cdot P$ pairs of μ and σ values for pin-to-output delay distributions for every input of every gate. Here K is the number of discretizations of the output load and P is the number of discretizations of the input slew values.

To the best of our knowledge, this is the first work which accelerates Monte Carlo based SSTA on a GPU platform. The key contributions of this work are as follows:

- We exploit the natural match between Monte Carlo based SSTA and the capabilities of a GPU, a SIMD-based device. We harness the tremendous computational power and memory bandwidth of GPUs to accelerate Monte Carlo based SSTA application.
- The implementation satisfies the key requirements to obtain maximal speedup on a GPU:
  - Different threads which generate normally distributed samples and perform STA computations are implemented so that there are no data dependencies between threads.
  - All gate evaluation threads compute identical instructions but on different data, which exploits the SIMD architecture of the GPU.
  - The μ and σ for the pin-to-output delay of any gate, required for a single STA computation, are obtained using a memory lookup, which exploits the extremely large memory bandwidth of GPUs.
- Our Monte Carlo based SSTA engine is implemented in a manner which is aware of the specific constraints of the GPU platform, such as the use of texture memory for table lookup, memory coalescing, use of shared memory, and use of a SIMD algorithm for generating random samples, thus maximizing the speedup obtained.
- Our implementation can obtain about 818× speedup compared to a CPU-based implementation. This includes the time required to transfer data to and from the GPU.
- Further, even though our current implementation has been benchmarked on a single NVIDIA GeForce GTX 280 graphics card, the NVIDIA SLI technology [7] supports up to four NVIDIA GeForce GTX 280 graphic cards on the same motherboard. We show that Monte Carlo based SSTA can be performed about 2,400×

faster on a quad GPU system, compared to a conventional single-core CPU-based implementation.

Our Monte Carlo based timing analysis is implemented in the Compute Unified Device Architecture (CUDA) framework [4, 3]. The GPU device used for our implementation and benchmarking is the NVIDIA GeForce 280 GTX. The correctness of our GPU-based timing analyzer has been verified by comparing its results with a CPU-based implementation of Monte Carlo based SSTA. An extended abstract of this work is available in [17].

7.3 Previous Work

The approaches of [11, 19] are some of the early works in SSTA. In recent times, the interest in this field has grown rapidly. This is primarily due to the fact that process variations are growing larger and less systematic, with shrinking feature sizes.

SSTA algorithms can be broadly categorized into *block based* and *path based*. In block-based algorithms, delay distributions are propagated by traversing the circuit under consideration in a levelized breadth-first manner. The fundamental operations in a block-based SSTA tool are the *SUM* and the *MAX* operations of the μ and σ values of the distributions. Therefore, block-based algorithms rely on efficient ways to implement these operations, rather than using discrete delay values. In path-based algorithms, a set of paths is selected for a detailed statistical analysis. While block-based algorithms [27, 20] tend to be fast, it is difficult to compute an accurate solution of the statistical MAX operation when dealing with correlated random variables or reconvergent fanouts. In such cases, only an approximation is computed, using the upper bound or lower bound of the probability distribution function (PDF) calculation or by using the moment matching technique [25]. The advantage of path-based methods is that they accurately calculate the delay PDF of each path since they do not rely on statistical MAX operations and can account for correlations between paths easily.

Similar to path-based SSTA approaches, our method does not need to perform statistical MAX and SUM operations. Our method is based on propagating the frontier of circuit delay values, obtained from the μ and σ values of the pin-to-output delay distributions for the gates in the design. Unlike path-based approaches, we do not need to select a set of paths to be analyzed.

The authors of [14] present a technique to propagate PDFs through a circuit in the same manner as arrival times of signals are propagated during STA. Principal component analysis enables them to handle spatial correlations of the process parameters. While the SUM of two Gaussian distributions yields another Gaussian distribution, the MAX of two or more Gaussian distributions is not a Gaussian distribution in general. As a simplification, and for ease of calculation, the authors of [14] approximate the MAX of two or more Gaussian distributions to be Gaussian as well.

A canonical first-order delay model is proposed in [12]. Based on this model, an incremental block-based timing analyzer is used to propagate arrival times and required times through a timing graph. In [10, 8, 9], the authors note that accurate SSTA can become exponential. Hence, they propose faster algorithms that compute only the bounds on the exact result.

In [15], a block based SSTA algorithm is discussed. By representing the arrival times as cumulative distribution functions and the gate delays as PDFs, the authors claim to have an efficient method to do the SUM and MAX operations. The accuracy of the algorithm can be adjusted by choosing more discretization levels. Reconvergent fanouts are handled through a statistical subtraction of the common mode. The authors of [21] propagate delay distributions through a circuit. The PDFs are discretized to help make the operation more efficient. The accuracy of the result in this case is again dependent on the discretization. The approach of [16] automates the process of false path removal implicitly (by using a sensitizable timing analysis methodology [24]). The approach first finds the primary input vector transitions that result in the sensitizable longest delays for the circuit and then performs a statistical analysis on these vector transitions alone.

In contrast to these approaches, our approach *accelerates* Monte Carlo based SSTA technique by using off-the-shelf commercial graphics processing units (GPUs). The ubiquity and ease of programming of GPU devices, along with their extremely low costs, makes GPUs an attractive choice for such an application.

7.4 Our Approach

We accelerate Monte Carlo based SSTA by implementing it on a graphics processing unit (GPU). The following sections describe the details of our implementation. Section 7.4.1 discusses the details of implementing STA on a GPU, while Section 7.4.2 extends this discussion for implementing SSTA on a GPU.

7.4.1 Static Timing Analysis (STA) at a Gate

The computation involved in a single STA evaluation at any gate of a design is as follows. At each gate, the MAX of the SUM of the input arrival time at pin i plus the pin-to-output rising (or falling) delay from pin i to the output is computed. The details are explained with the example of a NAND2 gate.

Consider a NAND2 gate. Let AT_i^{fall} denote the arrival time of a falling signal at node i and AT_i^{rise} denote the arrival time of a rising signal at node i. Let the two inputs of the NAND2 gate be a and b and the output be c.

The rising time (delay) at the output c of a NAND2 gate is calculated as shown below. A similar expression can be written to compute the falling delay at the output c:

$$AT_c^{\text{rise}} = \text{MAX}[(AT_a^{\text{fall}} + \text{MAX}(D_{11\rightarrow 00}, D_{11\rightarrow 01})), \\ (AT_b^{\text{fall}} + \text{MAX}(D_{11\rightarrow 00}, D_{11\rightarrow 10}))]$$

where, $\text{MAX}(D_{11\rightarrow 00}, D_{11\rightarrow 01})$ is the pin-to-output rising delay from the input a, while $\text{MAX}(D_{11\rightarrow 00}, D_{11\rightarrow 10})$ is the pin-to-output rising delay from the input b.

To implement the above computation on the GPU, a lookup table (LUT) based approach is employed. The pin-to-output rising and falling delay from every input for every gate is stored in a LUT. The output arrival time of an n-input gate G is then computed by calling the 2-input MAX operation $n-1$ times, after n computations of the SUM of the input arrival time plus the pin-to-output rising (or falling) gate delay. The pin-to-output delay for pin i is looked up in the LUT at an address corresponding to the base address of gate G and the offset for the transition on pin i. Since the LUT is typically small, these lookups are usually cached. Further, this technique is highly amenable to parallelization as will be shown in the sequel.

In our implementation of the LUT-based SSTA technique on a GPU, the LUTs (which contain the pin-to-output falling and rising delays) for all the gates are stored in the texture memory of the GPU device. This has the following advantages:

- Texture memory on a GPU device is cached unlike shared or global memory. Since the truth tables for all library gates easily fit into the available cache size, the cost of a lookup will typically be one clock cycle.
- Texture memory accesses do not have coalescing constraints as required for global memory accesses. This makes the gate lookup efficient.
- The latency of addressing calculations is better hidden, possibly improving performance for applications like STA that perform random accesses to the data.
- In case of multiple lookups performed in parallel, shared memory accesses might lead to bank conflicts and thus impede the potential improvement due to parallel computations.
- In the CUDA programming environment, there are built-in texture fetching routines which are extremely efficient.

The allocation and loading of the texture memory requires non-zero time, but is done only once for a library. This runtime cost is easily amortized since several STA computations are done, especially in an SSTA setting.

The GPU allows several threads to be active in parallel. Each thread in our implementation performs STA at a single n-input gate G by performing n lookups from the texture memory, n SUM operations, and $n-1$ MAX operations. The data, organized as a 'C' structure *type struct threadData*, is stored in the global memory of the device for all threads. The global memory, as discussed in Chapter 3, is accessible by all processors of all multiprocessors. Each processor executes multiple threads simultaneously. This organization thus requires multiple accesses to the global memory. Therefore, it is important that the memory coalescing constraint for a global memory access is satisfied. In other words, memory accesses should be performed in sizes equal to 32-bit, 64-bit, or 128-bit values. The data structure required by a thread for STA at a gate with four inputs is

typedef struct __align__(8){
int *offset*; // Gate type's offset
float *a*; float *b*; float *c*; float *d*; // input arrival times
} threadData;

The first line of the declaration defines the structure type and byte alignment (required for coalescing accesses). The elements of this structure are the offset in texture memory (type integer) of the gate, for which this thread will perform STA, and the input arrival times (type float).

The pseudocode of the kernel (the code executed by each thread) for the static timing analysis of an inverting gate (for a rising output) is given in Algorithm 5. The arguments to the routine *static_timing_kernel* are the pointers to the global memory for accessing the *threadData* (*MEM*) and the pointers to the global memory for storing the output delay value (*DEL*). The global memory is indexed at a location equal to the thread's unique *threadID* = t_x, and the *threadData* data for any gate is accessed from this base address in memory. Suppose the index of input x of the gate is i. Since we handle gates with up to 4 inputs, $0 \leq i \leq 3$. The pin-to-output rising (falling) delay for an input x of an *inverting* gate is accessed by indexing the LUT (in texture memory) at the sum of the gate's base address (*offset*) plus $2 \cdot i$ ($2 \cdot i$+1) for a falling (rising) transition. Similarly, the pin-to-output rising (falling) delay for an input x for a non-inverting gate is accessed by indexing the LUT (in texture memory) at the sum of the gate's base address (*offset*) plus $2 \cdot i$+1 ($2 \cdot i$) for a rising (falling) transition.

The CUDA inbuilt one-dimensional texture fetching function *tex*1*D*(*LUT*,*index*) is next invoked to fetch the corresponding pin-to-output delay values for every input. The fetched value is added to the input arrival time of the corresponding input. Then, using $n - 1$ MAX operations, the output arrival time is computed.

In our implementation, the same kernel implements gates with n = 1, 2, 3, or 4 inputs. For gates with less than four inputs, the extra memory in the LUT stores zeroes. This enables us to invoke the same kernel for any instance of a 2-, 3-, or 4-input inverting (non-inverting) gate.

Algorithm 5 Pseudocode of the Kernel for Rising Output STA for Inverting Gate

static_timing_kernel(*threadData* $*$ *MEM*,*float* $*$ *DEL*){
t_x = *my_thread_id*;
threadData Data = *MEM*[t_x];
p2pdelay_a = *tex*1*D*(*LUT*,*MEM*[t_x].*offset* + 2 × 0);
p2pdelay_b = *tex*1*D*(*LUT*,*MEM*[t_x].*offset* + 2 × 1);
p2pdelay_c = *tex*1*D*(*LUT*,*MEM*[t_x].*offset* + 2 × 2);
p2pdelay_d = *tex*1*D*(*LUT*,*MEM*[t_x].*offset* + 2 × 3);
LAT = *fmaxf*(*MEM*[t_x].*a* + *p2pdelay_a*,*MEM*[t_x].*b* + *p2pdelay_b*);
LAT = *fmaxf*(*LAT*,*MEM*[t_x].*c* + *p2pdelay_c*);
DEL[t_x] = *fmaxf*(*LAT*,*MEM*[t_x].*d* + *p2pdelay_d*);
}

7.4.2 Statistical Static Timing Analysis (SSTA) at a Gate

SSTA at a gate is performed by an implementation that is similar to the STA implementation discussed above. The additional information required is the μ and σ of the n Gaussian distributions of the pin-to-output delay values for the n inputs to the gate. The μ and σ used for each Gaussian distribution are stored in LUTs (as opposed to storing a simple nominal delay value as in the case of STA).

The pseudo-random number generator used for generating samples from the Gaussian distribution is the Mersenne Twister pseudo-random number generation algorithm [22]. It has many important properties like a long period, efficient use of memory, good distribution properties, and high performance.

As discussed in [5], the Mersenne Twister algorithm maps well onto the CUDA programming model. Further, a special offline library called *dcmt* (developed in [23]) is used for the dynamic creation of the Mersenne Twister parameters. Using *dcmt* prevents the creation of correlated sequences by threads that are issued in parallel.

Uniformly distributed random number sequences, produced by the Mersenne Twister algorithm, are then transformed into the normal distribution $N(0,1)$ using the Box–Muller transformation [1]. This transformation is implemented as a separate kernel.

The pseudocode of the kernel for the SSTA computations of an inverting gate (for the rising output) is given in Algorithm 6. The arguments to the routine *statistical_static_timing_kernel* are the pointers to the global memory for accessing the *threadData* (*MEM*) and the pointers to the global memory for storing the output delay value (*DEL*). The global memory is indexed at a location equal to the thread's unique $threadID = t_x$, and the *threadData* data of the gate is thus accessed. The μ and σ of the pin-to-output rising (falling) delay for an input x of an *inverting* gate are accessed by indexing LUT^{μ} and LUT^{σ}, respectively, at the sum of the gate's base address (*offset*) plus $2 \cdot i$ $(2 \cdot i+1)$ for a falling (rising) transition.

The CUDA inbuilt one-dimensional texture fetching function *tex*1*D*(*LUT*,*index*) is invoked to fetch the μ and σ corresponding to the pin-to-output delay's μ and σ values for every input. Using the pin-to-output μ and σ values, along with the Mersenne Twister pseudo-random number generator and the Box–Muller transformation, a normally distributed sample of the pin-to-output delay for every input is generated. This generated value is added to the input arrival time of the corresponding input. Then, by performing $n - 1$ MAX operations, the output arrival time is computed.

In our implementation of Monte Carlo based SSTA for a circuit, we first levelize the circuit. In other words, each gate of the netlist is assigned a level which is one more than the maximum level of its fanins. The primary inputs are assigned a level '0.' We then perform SSTA at all gates with level i, starting with i=1. Note that we do not store (on the GPU) the output arrival times for all the gates at any given time. We use the GPU's global memory for storing the arrival times of the gates in the current level that are being processed, along with their immediate fanins. We reclaim the memory used by all gates which are not inputs to any of the gates at the current or a higher level. By doing this we incur no loss of data since the entire

Algorithm 6 Pseudocode of the Kernel for Rising Output SSTA for Inverting Gate

```
statistical_static_timing_kernel(threadData * MEM,float * DEL){
t_x = my_thread_id;
threadData Data = MEM[t_x];
p2pdelay_a^μ = tex1D(LUT^μ,MEM[t_x].offset + 2 × 0);
p2pdelay_a^σ = tex1D(LUT^σ,MEM[t_x].offset + 2 × 0);
p2pdelay_b^μ = tex1D(LUT^μ,MEM[t_x].offset + 2 × 1);
p2pdelay_b^σ = tex1D(LUT^σ,MEM[t_x].offset + 2 × 1);
p2pdelay_c^μ = tex1D(LUT^μ,MEM[t_x].offset + 2 × 2);
p2pdelay_c^σ = tex1D(LUT^σ,MEM[t_x].offset + 2 × 2);
p2pdelay_d^μ = tex1D(LUT^μ,MEM[t_x].offset + 2 × 3);
p2pdelay_d^σ = tex1D(LUT^σ,MEM[t_x].offset + 2 × 3);
p2p_a = p2pdelay_a^μ + k_a × p2pdelay_a^σ ; // k_a, k_b, k_c, k_d
p2p_b = p2pdelay_b^μ + k_b × p2pdelay_b^σ ; // are obtained by Mersenne
p2p_c = p2pdelay_c^μ + k_c × p2pdelay_c^σ ; // Twister followed by
p2p_d = p2pdelay_d^μ + k_d × p2pdelay_d^σ ; // Box-Muller transformations.
LAT = fmaxf(MEM[t_x].a + p2p_a,MEM[t_x].b + p2p_b);
LAT = fmaxf(LAT,MEM[t_x].c + p2p_c);
DEL[t_x] = fmaxf(LAT,MEM[t_x].d + p2p_d);
}
```

approach is carried out in a single pass and we do not revisit any gate. Although our current implementation simultaneously simulates all gates with level i, the number of computations at each gate is large enough to keep the GPU's processors busy. Hence, we could alternatively simulate one gate at a time on the GPU. Therefore, our implementation poses no restrictions on the size of the circuit being processed.

GPUs allow extreme speedups if the different threads being evaluated have no data dependencies. The programming model of a GPU is the single instruction multiple data (SIMD) model, under which all threads must compute identical instructions, but on different data. Also, GPUs have an extremely large memory bandwidth, allowing multiple memory lookups to be performed in parallel.

Monte Carlo based SSTA requires multiple sample points for a single gate being analyzed. By exploiting sample parallelism, several sample points can be analyzed in parallel. Similarly, SSTA at each gate at a specific topological level in the circuit can be performed independently of SSTA at other gates. By exploiting this data parallelism, many gates can be analyzed in parallel. This maximally exploits the SIMD semantics of the GPU platform.

7.5 Experimental Results

In order to perform S gate evaluations for SSTA, we need to invoke S *statistical_static_timing_kernel*s in parallel. The total DRAM on an NVIDIA GeForce GTX 280 is 1 GB. This off-chip memory can be used as global, local, and texture memory. Also the same memory is used to store CUDA programs, context data used by the GPU device drivers, drivers for the desktop display, and NVIDIA control panels. The wall clock time taken for 16M executions of *statistical_static_timing_kernel*s

(by issuing 16M threads in parallel) is 0.023 s. A similar routine using the conventional implementation on a 3.6 GHz CPU with 3 GB RAM, running Linux, took 21.82 s for 16M calls. Thus asymptotically, the speedup of our implementation is $\sim$950×. The allocation and loading of the texture memory is a one time cost of about 0.18 ms, which is easily amortized in our implementation. Note that the Mersenne Twister implementation on the GTX 280, when compared to an implementation on the CPU (3.6 GHz CPU with 3 GB RAM), is by itself about 2 orders of magnitude faster. On the GTX 280, the Mersenne Twister kernel generates random numbers at the rate of 2.71 $\times 10^9$ numbers/s. A CPU implementation of the Mersenne Twister algorithm, on the other hand, generates random numbers at the rate of 1.47 $\times 10^7$ numbers/s. The results obtained from the GPU implementation were verified against the CPU results.

We ran 60 large IWLS, ITC, and ISCAS benchmark designs, to compute the per-circuit speed of our Monte Carlo based SSTA engine implemented on a GPU. These designs were first mapped in SIS [26] for delay optimality. The Monte Carlo analysis was performed with 1M samples. The results for 30 representative benchmark designs for our GPU-based SSTA approach are shown in Table 7.1. Column 1 lists the name of the circuit. Columns 2, 3, and 4 list the number of primary inputs, primary outputs, and gates in the circuit. Columns 5 and 7 list the GPU and CPU runtime, respectively. The time taken to transfer data between the CPU and GPU was accounted for in the GPU runtimes listed. In particular, the data transferred from the CPU to the GPU is the arrival times at each primary input and the μ and σ information for all pin-to-output delays of all gates. The data returned by the GPU are the 1M delay values at each output of the design. The runtimes also include the time required for the Mersenne Twister algorithm and the computation of the Box–Muller transformation. Column 8 reports the speedup obtained by using a single GPU card.

Using the NVIDIA SLI technology with four GPU chips on a single motherboard [7] allows for a 4× speedup in the processing time. The transfer times, however, do not scale. Column 6 lists the runtimes obtained when using a quad GPU system [7] and the corresponding speedup against the CPU implementation is reported in Column 9.

We also compared the performance of our Monte Carlo based SSTA approach (implemented on the GeForce 280 GTX) with a similar implementation on (i) Single-core and Dual-core Intel Conroe (Core 2) processors operating at 2.4 GHz, with 2 MB cache (implemented in a 65 nm technology) and (ii) Single-core, Dual-core, and Quad-core Intel Penryn (Core 2) processors operating at 3.0 GHz, with 3 MB cache (implemented in a 45 nm technology). The implementations on the Intel processors used the Intel Streaming SIMD Extensions (SSE) [2] instruction set which consists of 4-wide integer (and floating point) SIMD vector instructions. These comparisons were performed over 10 benchmarks. The normalized performance for all architectures is plotted in Fig. 7.1. The performance of the 280 GTX implementation of Monte Carlo based SSTA is on average 61× faster than Conroe (Single), the Intel Core 2 (single core) with SSE instructions.

Table 7.1 Monte Carlo based SSTA results

Circuit	# Inputs	# Outputs	# Gates	GPU runtimes (s)		CPU runtime (s)	Speedup	
				Single GPU	SLI Quad		Single GPU	SLI Quad
b14	276	299	9,496	19.39	5.85	17,263.73	890.54	2,949.15
b15_1	483	518	13,781	28.53	8.89	25,053.86	878.15	2,817.45
b17	1,450	1,511	41,174	85.24	26.57	74,854.33	878.17	2,817.63
b18	3,305	3,293	6,599	28.39	18.99	11,996.98	422.54	631.79
b21	521	512	20,977	42.35	12.46	38,136.19	900.50	3,061.26
b22_1	734	725	25,253	51.50	15.51	45,909.95	891.51	2,959.80
s832	23	24	587	1.23	0.39	1,067.17	870.09	2,736.15
s838.1	66	33	562	1.36	0.56	1,021.72	752.26	1,833.17
s1238	32	32	857	1.78	0.56	1,558.03	874.36	2,778.84
s1196	32	32	762	1.60	0.52	1,385.32	865.07	2,687.06
s1423	91	79	949	2.23	0.88	1,725.28	773.56	1,965.07
s1494	14	25	1,033	2.04	0.57	1,877.99	921.17	3,314.06
s1488	14	25	1,016	2.01	0.56	1,847.09	920.60	3,306.64
s5378	199	213	2,033	4.83	1.93	3,695.99	765.36	1,912.97
s9234.1	247	250	3,642	8.11	2.92	6,621.16	816.64	2,269.11
s13207	700	790	5,849	14.55	6.21	10,633.48	731.07	1,712.24
s15850	611	684	6,421	15.19	6.04	11,673.38	768.44	1,932.30
s35932	1,763	2,048	19,898	46.50	18.14	36,174.56	778.00	1,993.97
s38584	1,464	1,730	21,051	47.24	17.24	38,270.72	810.19	2,219.98
s38417	1,664	1,742	18,451	43.11	16.81	33,543.92	778.16	1,995.02
C1355	41	32	715	1.55	0.53	1,299.87	839.66	2,456.18
C1908	33	25	902	1.87	0.58	1,639.84	878.89	2,825.11
C2670	233	140	1,411	3.72	1.71	2,565.20	688.66	1,496.42
C3540	50	22	1,755	3.55	1.05	3,190.59	898.23	3,035.12
C432	36	7	317	0.75	0.30	576.31	766.47	1,919.90
C499	41	32	675	1.47	0.51	1,227.15	833.61	2,405.12
C5315	178	123	2,867	6.26	2.17	5,212.21	832.93	2,399.48
C6288	32	32	2,494	4.89	1.34	4,534.09	926.80	3,388.08
C7552	207	108	3,835	8.20	2.74	6,972.03	850.15	2,548.23
C880	60	26	486	1.18	0.49	883.55	746.11	1,797.11
Average							818.26	2,405.48

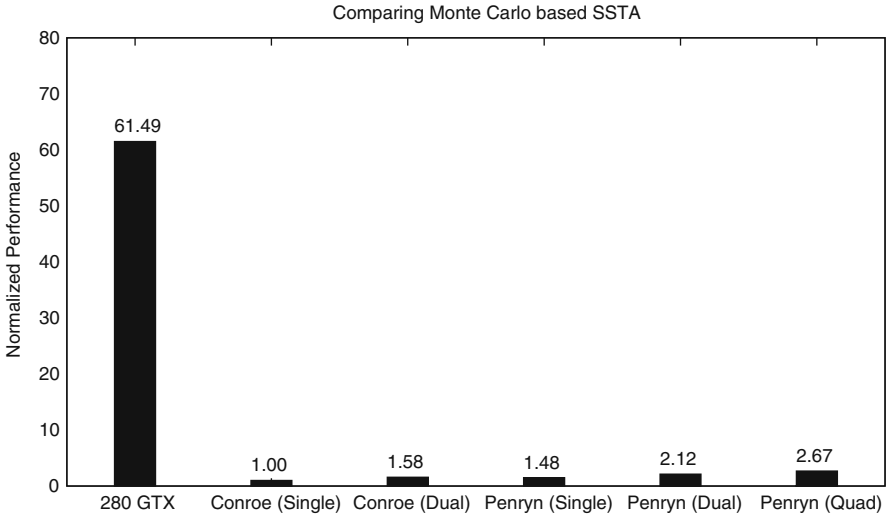


Fig. 7.1 Comparing Monte Carlo based SSTA on GTX 280 GPU and Intel Core 2 processors (with SEE instructions)

7.6 Chapter Summary

In this chapter, we have presented the implementation of Monte Carlo based SSTA on a graphics processing unit. Monte Carlo based SSTA is computationally expensive, but crucial for design timing closure since it enables an accurate analysis of the delay variations. Our implementation computes multiple timing analysis evaluations of a single gate in parallel. We used a SIMD implementation of the Mersenne Twister pseudo-random number generator, followed by Box–Muller transformations, (both implemented on the GPU) for generating delay numbers in a normal distribution. The μ and σ of the pin-to-output delay numbers, for all inputs and for every gate, are obtained using a memory lookup, which exploits the large memory bandwidth of the GPU. Threads which execute in parallel do not have data or control dependencies. All threads execute identical instructions, but on different data. This is in accordance with the SIMD programming semantics of the GPU. Our results, implemented on an NVIDIA GeForce GTX 280 GPU card, indicate that our approach can provide about 818$\times$ speedup when compared to a conventional CPU implementation. With the quad 280 GPU cards [7], our projected speedup is $\sim$2,400$\times$.

References

1. Box–Muller Transformation. `http://mathworld.wolfram.com/Box-Muller Transformation`
2. Intel SSE. `http://www.tommesani.com/SSE.html`

3. NVIDIA CUDA Homepage. http://developer.nvidia.com/object/cuda.html
4. NVIDIA CUDA Introduction. http://www.beyond3d.com/content/articles/12/1
5. Parallel Mersenne Twister. http://developer.download.nvidia.com/~MersenneTwister
6. SLI Technology. http://www.slizone.com/page/slizone.html
7. SLI Technology. http://www.slizone.com/page/slizone.html
8. Agarwal, A., Blaauw, D., Zolotov, V.: Statistical timing analysis for intra-die process variations with spatial correlations. In: Proceedings of the International Conference on Computer-Aided Design, pp. 900–907 (2003)
9. Agarwal, A., Blaauw, D., Zolotov, V., Vrudhula, S.: Statistical timing analysis using bounds. In: Proceedings of the Conference on Design Automation and Test in Europe, pp. 62–67 (2003)
10. Agarwal, A., Zolotov, V., Blaauw, D.T.: Statistical timing analysis using bounds and selective enumeration. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, pp. 1243–1260 (2003)
11. Benkoski, J., Strojwas, A.J.: A new approach to hierarchical and statistical timing simulations. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 6, pp. 1039–1052 (1987)
12. C, C.V., Ravindran, K., Kalafala, K., Walker, S.G., Narayan, S.: First-order incremental block-based statistical timing analysis. In: Proceedings of the Design Automation Conference, pp. 331–336 (2004)
13. Cabaleiro, J., Carazo, J., Zapata, E.: Parallel algorithm for principal component analysis based on Hotelling procedure. In: Proceedings of EUROMICRO Workshop On Parallel and Distributed Processing, pp. 144–149 (1993)
14. Chang, H., Sapatnekar, S.S.: Statistical timing analysis under spatial correlations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**(9), 1467–1482 (2005)
15. Devgan, A., Kashyap, C.V.: Block-based static timing analysis with uncertainty. In: Proceedings of the International Conference on Computer-Aided Design, pp. 607–614 (2003)
16. Garg, R., Jayakumar, N., Khatri, S.P.: On the improvement of statistical timing analysis. International Conference on Computer Design pp. 37–42 (2006)
17. Gulati, K., Khatri, S.P.: Accelerating statistical static timing analysis using graphics processing units. In: Proceedings, IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC), pp. 260–265 (2009)
18. Heras, D., Cabaleiro, J., Perez, V., Costas, P., Rivera, F.: Principal component analysis on vector computers. In: Proceedings of VECPAR, pp. 416–428 (1996)
19. Jyu, H.F., Malik, S.: Statistical delay modeling in logic design and synthesis. In: Proceedings of the Design Automation Conference, pp. 126–130 (1994)
20. Le, J., Li, X., Pileggi, L.T.: STAC: Statistical timing analysis with correlation. In: DAC '04: Proceedings of the 41st annual conference on Design automation, pp. 343–348 (2004)
21. Liou, J.J., Cheng, K.T., Kundu, S., Krstic, A.: Fast statistical timing analysis by probabilistic event propagation. In: Proceedings of the Design Automation Conference, pp. 661–666 (2001)
22. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling, and Computer Simulation **8**(1), 3–30 (1998)
23. Matsumoto, M., Nishimura, T.: Monte Carlo and Quasi-Monte Carlo Methods, chap. Dynamic Creation of Pseudorandom Number Generators, pp. 56–69. Springer, New York (1998)
24. McGeer, P., Saldanha, A., Brayton, R., Sangiovanni-Vincentelli, A.: Logic Synthesis and Optimization, chap. Delay Models and Exact Timing Analysis, pp. 167–189. Kluwer Academic Publishers, Dordrecht (1993)
25. Nitta, I., Shibuya, T., Homma, K.: Statistical static timing analysis technology. FUJITSU Scientific and Technical Journal **43**(4), 516–523 (2007)

26. Sentovich, E.M., Singh, K.J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P.R., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, CA 94720 (1992)
27. Zhang, L., Chen, W., Hu, Y., Chen, C.C.P.: Statistical timing analysis with extended pseudo-canonical timing model. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pp. 952–957 (2005)

Chapter 8
Accelerating Fault Simulation Using Graphics Processors

8.1 Chapter Overview

In this chapter, we explore the implementation of fault simulation on a graphics processing unit (GPU). In particular, we implement a parallel fault simulator. Fault simulation is inherently parallelizable, and the large number of threads that can be computed in parallel on a GPU results in a natural fit for the problem of parallel fault simulation. Our implementation fault-simulates all the gates in a particular level of a circuit, including good- and faulty-circuit simulations, for all patterns, in parallel. Since GPUs have an extremely large memory bandwidth, we implement each of our fault simulation threads (which execute in parallel with no data dependencies) using memory lookup. Fault injection is also done along with gate evaluation, with each thread using a different fault injection mask. All threads compute identical instructions, but on different data, as required by the single instruction multiple data (SIMD) programming semantics of the GPU. Our results, implemented on an NVIDIA GeForce GTX 280 GPU card, indicate that our approach is on average 47× faster when compared to a commercial fault simulation engine. With the NVIDIA Tesla cards (which can house eight 280 GTX GPU cards) our approach would be potentially 300× faster. The correctness of the GPU-based fault simulator has been verified by comparing its result with a CPU-based fault simulator.

The remainder of this chapter is organized as follows: Section 8.2 discusses the motivation to accelerate fault simulation. Some previous work in fault simulation has been described in Section 8.3. Section 8.4 details our approach for implementing LUT-based fault simulation on GPUs. In Section 8.5 we present results from experiments which were conducted in order to benchmark our approach. We summarize the chapter in Section 8.6.

8.2 Introduction

Fault simulation is an important step of the VLSI design flow. Given a digital design and a set of input vectors V defined over its primary inputs, fault simulation evaluates the number of stuck-at faults F_{sim} that are tested by applying the vectors V. The

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_8,
© Springer Science+Business Media, LLC 2010

ratio of F_{sim} to the total number of faults in the design F_{total} is a measure of the fault coverage. The task of finding this ratio is often referred to as *fault grading* in the industry. For today's complex digital designs with N logic gates (N is often in several million), the number of faulty variations of the design can be dramatically higher. Therefore, it is extremely important to explore ways to accelerate fault simulation. The ideal fault simulation approach should be fast, scalable, *and* cost effective.

Parallel processing of fault simulation computations is an approach that has routinely been invoked to reduce the compute time of fault simulation [8]. Fault simulation can be parallelized by a variety of techniques. The techniques include parallelizing the fault simulation algorithm (*algorithm-parallel techniques* [6, 4, 5]), partitioning the circuit into disjoint components and simulating them in parallel (*model-parallel* techniques [13, 20]), partitioning the fault set data and simulating faults in parallel (*data-parallel* techniques [18, 9, 15, 14, 19, 11, 7]), and a combination of one or more of these techniques [12]. Data-parallel techniques can be further classified into *fault-parallel* methods, wherein different faults are simulated in parallel, and *pattern-parallel* approaches, wherein different patterns of the same fault are simulated in parallel. In this chapter, we present an accelerated fault simulation approach that invokes data parallelism. In particular, *both* fault and pattern parallelism are exploited by our method. The method is implemented on a graphics processing unit (GPU) platform.

Fault simulation of a logic netlist effectively requires multiple logic simulations of the netlist, with faults injected at various gates (typically primary inputs and reconvergent fanout branches). An approach for logic simulation (which can also be used for fault simulation) uses lookup table (LUT) based computations. In this approach the truth table for all the gates in a library is stored in the memory, and multiple processors perform multiple gate-level (logic) simulations in parallel. This is a natural match for the GPU capabilities, since it exploits the extremely high memory bandwidths of the GPU and also simultaneously utilizes the large number of computational elements on the GPU. Several faults (and several patterns for these faults) can be simulated simultaneously. In this way, both data parallelism and pattern parallelism are employed. The key point to note is that the *same* operation (of looking up gate output values in the memory) is performed on independent data (different faults and different patterns for every fault). In this way, the SIMD computing paradigm of the GPU is exploited maximally by fault simulation computations that are LUT based.

This work is the first approach, to the best of the authors' knowledge, which accelerates fault simulation on a GPU platform. The key contributions of this work are as follows:

- We exploit the novel match between data- and pattern-parallel fault simulation with the capabilities of a GPU (a SIMD-based device) and harness the computational power of GPUs to accelerate parallel fault simulation.
- The implementation satisfies all the key requirements which ensure maximal speedup in a GPU:

  - The different threads, which perform gate evaluations and fault injection, are implemented so that there are no data dependencies between threads.
  - All gate evaluation threads compute identical instructions, but on different data, which exploits the SIMD architecture of the GPU.
  - The gate evaluation is done using a LUT, which exploits the extremely large memory bandwidth of GPUs.

- Our parallel fault simulation algorithm is implemented in a manner which is aware of the specific constraints of the GPU platform, such as the use of texture memory for table lookup, memory coalescing, and use of shared memory, thus maximizing the speedup obtained.
- In comparison to a commercial fault simulation tool [1] our implementation is on average $\sim 47\times$ faster for fault simulating 32K patterns for each of 25 IWLS benchmarks [2].
- Further, even though our current implementation has been benchmarked on a single NVIDIA GeForce GTX 280 graphics card, the commercially available NVIDIA Tesla cards [3] allow up to eight NVIDIA GeForce GTX 280 devices on the same motherboard. We project that our implementation, on a Tesla card, performs fault simulation on average $\sim 300\times$ faster, when compared to the commercial tool.

Our fault simulation algorithm is implemented in the Compute Unified Device Architecture (CUDA), which is an open-source programming and interfacing tool provided by NVIDIA corporation, for programming NVIDIA's GPU devices. The GPU device used for our implementation and benchmarking is NVIDIA GTX 280 GPU card. The correctness of our GPU-based fault simulator has been verified by comparing its results with a CPU-based serial fault simulator. An extended abstract of this work is available in [10].

8.3 Previous Work

Over the last three decades, several research efforts have attempted to accelerate the problem of fault simulation in a scalable and cost-effective fashion, by exploiting the parallelism inherent in the problem These efforts can be divided into *algorithm parallel*, *model parallel*, and *data parallel*.

Algorithm-parallel efforts aim at parallelizing the fault simulation algorithm, distributing workload, and/or pipelining the tasks, such that the frequency of communication and synchronization between processors is reduced [12, 6, 4, 5]. In contrast to these approaches, our approach is data parallel. In [12], the authors aim at heuristically assigning fault set partitions (and corresponding circuit partitions) to several medium-grain multiprocessors. This assignment is based on a performance model developed by comparing the communication (message passing or shared memory access) to computation ratio of the multiprocessor units. The results reported in [12] are based on an implementation of fault simulation on a multiprocessor prototype

with up to eight processing units. Our results, on the other hand, are based on off-the-shelf GPU cards (the NVIDIA GeForce GTX 280 GPU). The authors of [6] present a methodology to predict and characterize workload distribution, which can aid in parallelizing fault simulation. The approach discussed in [4] suggests a pipelined design, where each functional unit performs a specific task. MARS [5], a hardware accelerator, is based on this design. However, the application of the accelerator to fault simulation has been limited [12].

In a model-parallel approach [13, 20, 12], the circuit to be simulated is partitioned into several (possibly non-disjoint) components. Each component is assigned to one or more processors. Further, in order to keep the partitioning balanced, dynamic re-partitioning [16, 17] is performed. This increases algorithm complexity and may impact simulation time [16, 17].

Numerous data-parallel approaches for fault simulation have been developed in the past. These approaches use dedicated hardware accelerators, supercomputers, vector machines, or multiprocessors [18, 9, 15, 14, 19, 11, 7]. There are several hardware-accelerated fault simulators in the literature, but they require specialized hardware, significant design effort and time, and non-trivial algorithm and software design efforts as well. In contrast to these approaches, our approach accelerates fault simulation by using off-the-shelf commercial graphics processing units (GPUs). The ubiquity and ease of programming of GPU devices, along with their extremely low costs compared to hardware accelerators, supercomputers, etc., make GPUs an attractive alternative for fault simulation.

8.4 Our Approach

GPUs allow extreme speedups if the different threads being evaluated have no data dependencies. The programming model of a GPU is the single instruction multiple data (SIMD) model, under which all threads must compute identical instructions, but on different data. Also, GPUs have an extremely large memory bandwidth, allowing multiple memory lookups to be performed in parallel.

Since fault simulation requires multiple (faulty) copies of the same circuit to be simulated, it forms a natural match to the capabilities of the GPU. Also, each gate evaluation *within a specific level in the circuit* can be performed independently of other gate evaluations. As a result, if we perform each gate evaluation (for gates with the same topological level) on a separate GPU thread, these threads will naturally satisfy the condition required for speedup in the GPU (which requires that threads have no data dependencies). Also, we implement fault simulation on the GPU, which allows each of the gate evaluations in a fault simulator to utilize the same thread code, with no conditional computations between or within threads. In particular, we implement pattern-parallel and fault-parallel fault simulation. Fault injection is also done along with gate evaluation, with each thread using a different fault injection mask. This maximally exploits the SIMD computing semantics of the GPU platform. Finally, in order to exploit the extreme memory bandwidths

offered by GPUs, our implementation of the gate evaluation thread uses a memory lookup-based logic simulation paradigm.

Fault simulation of a logic netlist consists of multiple logic simulations of the netlist with faults injected on specific nets. In the next three subsections we discuss (i) GPU-based implementation of logic simulation at a gate, (ii) fault injection at a gate, and (iii) fault detection at a gate. Then we discuss (iv) the implementation of fault simulation for a circuit. This uses the implementations described in the first three subsections.

8.4.1 Logic Simulation at a Gate

Logic simulation on the GPU is implemented using a lookup table (LUT) based approach. In this approach, the truth tables of all gates in the library are stored in a LUT. The output of the simulation of a gate of type G is computed by looking up the LUT at the address corresponding to the sum of the gate offset of G (G_{off}) and the value of the gate inputs.

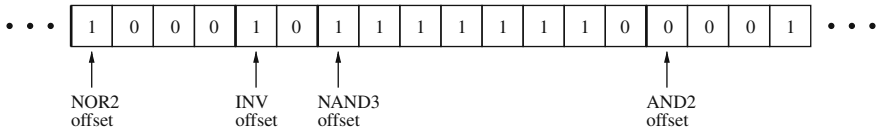


Fig. 8.1 Truth tables stored in a lookup table

Figure 8.1 shows the truth tables for a single NOR2, INV, NAND3, and AND2 gate stored in a one-dimensional lookup table. Consider a gate g of type NAND3 with inputs A, B, and C and output O. For instance if ABC = '110,' O should be '1.' In this case, logic simulation is performed by reading the value stored in the LUT at the address $\text{NAND3}_{\text{off}} + 6$. Thus, the value returned from the LUT will be the value of the output of the gate being simulated, for the particular input value. LUT-based simulation is a fast technique, even when used on a serial processor, since any gate (including complex gates) can be evaluated by a single lookup. Since the LUT is typically small, these lookups are usually cached. Further, this technique is highly amenable to parallelization as will be shown in the sequel. Note that in our implementation, each LUT enables the simulation of two identical gates (with possibly different inputs) simultaneously.

In our implementation of the LUT-based logic simulation technique on a GPU, the truth tables for all the gates are stored in the texture memory of the GPU device. This has the following advantages:

- Texture memory of a GPU device is cached as opposed to shared or global memory. Since the truth tables for all library gates will typically fit into the available cache size, the cost of a lookup will be one cycle (which is 8,192 bytes per multiprocessor).

- Texture memory accesses do not have coalescing constraints as required in case of global memory accesses, making the gate lookup efficient.
- In case of multiple lookups performed in parallel, shared memory accesses might lead to bank conflicts and thus impede the potential improvement due to parallel computations.
- Constant memory accesses in the GPU are optimal when all lookups occur at the same memory location. This is typically not the case in parallel logic simulation.
- The latency of addressing calculations is better hidden, possibly improving performance for applications like fault simulation that perform random accesses to the data.
- The CUDA programming environment has built-in texture fetching routines which are extremely efficient.

Note that the allocation and loading of the texture memory requires non-zero time, but is done only once for a gate library. This runtime cost is easily amortized since several million lookups are typically performed on a given design (with the same library).

The GPU allows several threads to be active in parallel. Each thread in our implementation performs logic simulation of two gates of the same type (with possibly different input values) by performing a single lookup from the texture memory.

The data required by each thread is the offset of the gate type in the texture memory and the input values of the two gates. For example, if the first gate has a 1 value for some input, while the second gate has a 0 value for the same input, then the input to the thread evaluating these two gates is '10.' In general, any input will have values from the set {00, 01, 10, 11}, or equivalently an integer in the range [0,3]. A 2-input gate therefore has 16 entries in the LUT, while a 3-input gate has 64 entries. Each entry of the LUT is a word, which provides the output for both the gates. Our gate library consists of an inverter as well as 2-, 3-, and 4-input NAND, NOR, AND, and OR gates. As a result, the total LUT size is $4+4\times(16+64+256) = 1{,}348$ words. Hence the LUT fits in the texture cache (which is 8,192 bytes per multiprocessor). Simulating more than two gates simultaneously per thread does not allow the LUT to fit in the texture cache, hence we only simulate two gates simultaneously per thread.

The data required by each thread is organized as a 'C' structure *type struct threadData* and is stored in the global memory of the device for all threads. The global memory, as discussed in Chapter 3, is accessible by all processors of all multiprocessors. Each processor executes multiple threads simultaneously. This organization would thus require multiple accesses to the global memory. Therefore, it is important that the memory coalescing constraint for a global memory access is satisfied. In other words, memory accesses should be performed in sizes equal to 32-bit, 64-bit, or 128-bit values. In our implementation the *threadData* is *aligned* at 128-bit (= 16 byte) boundaries to satisfy this constraint. The data structure required by a thread for simultaneous logic simulation of a pair of identical gates with up to four inputs is

typedef struct __align__(16){
int *offset*; // Gate type's offset
int *a*; int *b*; int *c*; int *d*;// input values
int m_0; int m_1; // fault injection bits
} threadData;

The first line of the declaration defines the structure type and byte alignment (required for coalescing accesses). The elements of this structure are the offset in texture memory (type integer) of the gate which this thread will simulate, the input signal values (type integer), and variables m_0 and m_1 (type integer). Variables m_0 and m_1 are required for fault injection and will be explained in the next subsection. Note that the total memory required for each of these structures is 1×4 bytes for the offset of type int + 4×4 bytes for the 4 inputs of type integer and 2×4 bytes for the fault injection bits of type integer. The total storage is thus 28 bytes, which is aligned to a 16 byte boundary, thus requiring 32 byte coalesced reads.

The pseudocode of the kernel (the code executed by each thread) for logic simulation is given in Algorithm 7. The arguments to the routine *logic_simulation_kernel* are the pointers to the global memory for accessing the *threadData* (*MEM*) and the pointer to the global memory for storing the output value of the simulation (*RES*). The global memory is indexed at a location equal to the thread's unique *threadID* = t_x, and the *threadData* data is accessed. The index I to be fetched in the LUT (in texture memory) is then computed by summing the gate's offset and the decimal sum of the input values for each of the gates being simultaneously simulated. Recall that each input value $\in \{0, 1, 2, 3\}$, representing the inputs of both the gates. The CUDA inbuilt single-dimension texture fetching function *tex*1*D*(LUT,*I*) is next invoked to fetch the output values of both gates. This is written at the t_x location of the output memory *RES*.

Algorithm 7 Pseudocode of the Kernel for Logic Simulation

logic_simulation_kernel(*threadData* $*$*MEM*, *int* $*$*RES*){
t_x = *my_thread_id*
threadData Data = *MEM*[t_x]
$I = Data.offset + 4^0 \times Data.a + 4^1 \times Data.b + 4^2 \times Data.c + 4^3 \times Data.d$
int output = *tex*1*D*(*LUT*,*I*)
RES[t_x] = *output*
}

8.4.2 Fault Injection at a Gate

In order to simulate faulty copies of a netlist, faults have to be *injected* at appropriate positions in the copies of the original netlist. This is performed by masking the appropriate simulation values by using a fault injection mask.

Our implementation parallelizes fault injection by performing a masking operation on the output value generated by the lookup (Algorithm 7). This masked value is now returned in the output memory *RES*. Each thread has it own masking bits m_0 and m_1, as shown in the *threadData* structure. The encoding of these bits are tabulated in Table 8.1.

Table 8.1 Encoding of the mask bits

m_0	m_1	Meaning
–	11	Stuck-at-1 mask
11	00	No fault injection
00	00	Stuck-at-0 mask

The pseudocode of the kernel to perform logic simulation followed by fault injection is identical to pseudocode for logic simulation (Algorithm 1) except for the last line which is modified to read

$RES[t_x] = (output \ \& \ Data.m_0) \parallel Data.m_1$

$RES[t_x]$ is thus appropriately masked for stuck-at-0, stuck-at-1, or no injected fault. Note that the two gates being simulated in the thread correspond to the same gate of the circuit, simulated for different patterns. The kernel which executes logic simulation followed by fault injection is called *fault_simulation_kernel*.

8.4.3 Fault Detection at a Gate

For an applied vector at the primary inputs (PIs), in order for a fault f to be detected at a primary output gate g, the good-circuit simulation value of g should be different from the value obtained by faulty-circuit simulation at g, for the fault f.

In our implementation, the comparison between the output of a thread that is simulating a gate driving a circuit primary output and the good-circuit value of this primary output is performed as follows. The modified *threadData_Detect* structure and the pseudocode of the kernel for fault detection (Algorithm 8) are shown below:

```
typedef struct __align__(16) {
int offset; // Gate type's offset
int a; int b; int c; int d;// input values
int Good_Circuit_threadID; // The thread ID which computes
//the Good circuit simulation
} threadData_Detect;
```

The pseudocode of the kernel for fault detection is shown in Algorithm 8. This kernel is only run for the primary outputs of the design. The arguments to the routine *fault_detection_kernel* are the pointers to the global memory for accessing the *threadData_Detect* structure (*MEM*), a pointer to the global memory for storing the output value of the good-circuit simulation (*GoodSim*), and a pointer in memory (*faultindex*) to store a 1 if the simulation performed in the thread results in fault detection (*Detect*). The first four lines of Algorithm 8 are identical to those of Algorithm 7. Next, a thread computing the good-circuit simulation value will

Algorithm 8 Pseudocode of the Kernel for Fault Detection

```
fault_detection_kernel(threadData_Detect *MEM, int *GoodSim, int *Detect,int *faultindex){
t_x = my_thread_id
threadData_Detect Data = MEM[t_x]
I = Data.offset + 4^0 × Data.a + 4^1 × Data.b + 4^2 × Data.c + 4^3 × Data.d
int output = tex1D(LUT,I)
if (t_x == Data.Good_Circuit_threadID) then
    GoodSim[t_x] = output
end if
__synch_threads()
Detect[faultindex] = ((output ⊕ GoodSim[Data.Good_Circuit_threadID])?1:0)
}
```

write its output to global memory. Such a thread will have its *threadID* identical to the Data.Good_Circuit_threadID. At this point a thread synchronizing routine, provided by CUDA, is invoked. If more than one good-circuit simulation (for more than one pattern) is performed simultaneously, the completion of all the writes to the global memory has to be ensured before proceeding. The thread synchronizing routine guarantees this. Once all threads in a block have reached the point where this routine is invoked, kernel execution resumes normally. Now all threads, including the thread which performed the good-circuit simulation, will read the location in the global memory which corresponds to its good-circuit simulation value. Thus, by ensuring the completeness of the writes prior to the reads, the thread synchronizing routine avoids write-after-read (WAR) hazards. Next, all threads compare the output of the logic simulation performed by them to the value of the good-circuit simulation. If these values are different, then the thread will write a 1 to a location indexed by its *faultindex*, in *Detect*, else it will write a 0 to this location. At this point the host can copy the *Detect* portion of the device global memory back to the CPU. All faults listed in the *Detect* vector are detected.

8.4.4 Fault Simulation of a Circuit

Our GPU-based fault simulation methodology is parallelized using the two data-parallel techniques, namely fault parallelism and pattern parallelism. Given the large number of threads that can be executed in parallel on a GPU, we use both these forms of parallelism simultaneously. This section describes the implementation of this two-way parallelism.

Given a logic netlist, we first levelize the circuit. By levelization we mean that each gate of the netlist is assigned a level which is one more than the maximum level of its input gates. The primary inputs are assigned a level '0.' Thus, Level(G) = max($\forall_{i \in \text{fanin}(G)}$Level($i$)) + 1. The maximum number of levels in a circuit is referred to as L. The number of gates at a level i is referred to as W_i. The maximum number of gates at any level is referred to as $W_{\max}$, i.e., ($W_{\max}$ = max($\forall_i(W_i)$)). Figure 8.2 shows a logic netlist with primary inputs on the extreme left and primary outputs

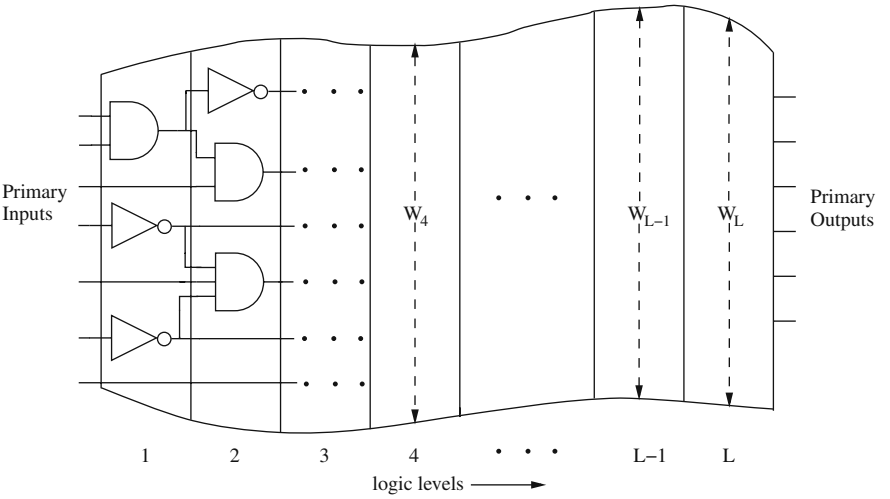


Fig. 8.2 Levelized logic netlist

on the extreme right. The netlist has been levelized and the number of gates at any level i is labeled W_i. We perform data-parallel fault simulation on all logic gates in a single level simultaneously.

Suppose there are N vectors (patterns) to be fault simulated for the circuit. Our fault simulation engine first computes the good-circuit values for all gates, for all N patterns. This information is then transferred back to the CPU, which therefore has the good-circuit values at each gate for each pattern. In the second phase, the CPU schedules the gate evaluations for the fault simulation of each fault. This is done by calling (i) *fault_simulation_kernel* (with fault injection) for each faulty gate G, (ii) the same *fault_simulation_kernel* (but without fault injection) on gates in the transitive fanout (TFO) of G, and (iii) *fault_detection_kernel* for the primary outputs in the TFO of G.

We reduce the number of fault simulations by making use of the good-circuit values of each gate for each pattern. Recall that this information was returned to the CPU after the first phase. For any gate G, if its good-circuit value is v for pattern p, then fault simulation for the stuck-at-v value on G is not scheduled in the second phase. In our experiments, the results include the time spent for the data transfers from CPU $\leftrightarrow$ GPU in all phases of the operation of out fault simulation engine. GPU runtimes also include all the time spent by the CPU to schedule good/faulty gate evaluations.

A few key observations are made at this juncture:

- Data-parallel fault simulation is performed on all gates of a level i simultaneously.
- Pattern-parallel fault simulation is performed on N patterns for any gate simultaneously.
- For all levels other than the last level, we invoke the kernel *fault_simulation_kernel*. For the last level we invoke the kernel *fault_detection_kernel*.

- Note that no limit is imposed by the GPU on the size of the circuit, since the entire circuit is never statically stored in GPU memory.

8.5 Experimental Results

In order to perform T_S logic simulations plus fault injections in parallel, we need to invoke T_S *fault_simulation_kernel*s in parallel. The total DRAM (off-chip) in the NVIDIA GeForce GTX 280 is 1 GB. This off-chip memory can be used as global, local, and texture memory. Also the same memory is used to store CUDA programs, context data used by the GPU device drivers, drivers for the desktop display, and NVIDIA control panels. With the remaining memory, we can invoke T_S = 32M *fault_simulation_kernel*s in parallel. The time taken for 32M *fault_simulation_kernel*s is 85.398 ms. The time taken for 32M *fault_detection_kernel*s is 180.440 ms.

The fault simulation results obtained from the GPU implementation were verified against a CPU-based serial fault simulator and were found to verify with 100% fidelity.

We ran 25 large IWLS benchmark [2] designs, to compute the speed of our GPU-based parallel fault simulation tool. We fault-simulated 32K patterns for all circuits. We compared our runtimes with those obtained using a commercial fault simulation tool [1]. The commercial tool was run on a 1.5 GHz UltraSPARC-IV+ processor with 1.6 GB of RAM, running Solaris 9.

The results for our GPU-based fault simulation tool are shown in Table 8.2. Column 1 lists the name of the circuit. Column 2 lists the number of gates in the mapped circuit. Columns 3 and 4 list the number of primary inputs and outputs for these circuits. The number of collapsed faults F_{total} in the circuit is listed in Column 5. These values were computed using the commercial tool. Columns 6 and 7 list the runtimes, in seconds, for simulating 32K patterns, using the commercial tool and our implementation, respectively. The time taken to transfer data between the CPU and GPU was accounted for in the GPU runtimes listed. In particular, the data transferred from the CPU to the GPU is the 32 K patterns at the primary inputs and the truth table for all gates in the library. The data transferred from GPU to CPU is the array *Detect* (which is of type Boolean and has length equal to the number of faults in the circuit). The commercial tool's runtimes include the time taken to read the circuit netlist and 32K patterns. The speedup obtained using a single GPU card is listed in Column 9.

By using the NVIDIA Tesla server housing up to eight GPUs [3], the available global memory increases by 8×. Hence we can potentially launch 8× more threads simultaneously. This allows for a 8× speedup in the processing time. However, the transfer times do not scale. Column 8 lists the runtimes on a Tesla GPU system. The speedup obtained against the commercial tool in this case is listed in Column 10. Our results indicate that our approach, implemented on a single NVIDIA GeForce GTX 280 GPU card, can perform fault simulation on average 47× faster when

Table 8.2 Parallel fault simulation results

					Runtimes (in seconds)			Speedup	
Circuit	# Gates	# Inputs	# Outputs	# Faults	Comm. tool	Single GPU	Tesla	Single GPU	Tesla
s9234_1	1,462	38	39	3,883	6.190	0.134	0.022	46.067	275.754
s832	417	20	19	937	3.140	0.031	0.005	101.557	672.071
s820	430	20	19	955	3.060	0.032	0.005	95.515	635.921
s713	299	37	23	624	4.300	0.029	0.005	146.951	883.196
s641	297	37	23	610	4.260	0.029	0.005	144.821	871.541
s5378	1,907	37	49	4,821	8.390	0.155	0.025	54.052	333.344
s38584	12,068	14	278	30,989	38.310	0.984	0.177	38.940	216.430
s38417	15,647	30	106	36,235	17.970	1.405	0.254	12.788	70.711
s35932	14,828	37	320	34,628	51.920	1.390	0.260	37.352	199.723
s15850	1,202	16	87	3,006	9.910	0.133	0.024	74.571	421.137
s1494	830	10	19	1,790	3.020	0.049	0.007	62.002	434.315
s1488	818	10	19	1,760	2.980	0.048	0.007	61.714	431.827
s13207	2,195	33	121	5,735	14.980	0.260	0.047	57.648	320.997
s1238	761	16	14	1,739	2.750	0.049	0.007	56.393	385.502
s1196	641	16	14	1,506	2.620	0.044	0.007	59.315	392.533
b22_1	34,985	34	22	86,052	16.530	1.514	0.225	10.917	73.423
b22	35,280	34	22	86,205	17.130	1.504	0.225	11.390	75.970
b21	22,963	34	22	56,870	11.960	1.208	0.177	9.897	67.656
b20_1	23,340	34	22	58,742	11.980	1.206	0.176	9.931	68.117
b20	23,727	34	22	58,649	11.940	1.206	0.177	9.898	67.648
b18	136,517	38	23	332,927	59.850	5.210	0.676	11.488	88.483
b15_1	17,510	38	70	43,770	16.910	0.931	0.141	18.166	119.995
b15	17,540	38	70	43,956	17.950	0.943	0.143	19.035	125.916
b14_1	10,640	34	54	26,494	11.530	0.641	0.093	17.977	123.783
b14	10,582	34	54	26,024	11.520	0.637	0.093	18.082	124.389
Average								47.459	299.215

compared to the commercial fault simulation tool [1]. With the NVIDIA Tesla card, our approach would be potentially 300× faster.

8.6 Chapter Summary

In this chapter, we have presented our implementation of a fault simulation engine on a graphics processing unit (GPU). Fault simulation is inherently parallelizable, and the large number of threads that can be computed in parallel on a GPU can be employed to perform a large number of gate evaluations in parallel. As a consequence, the GPU platform is a natural candidate for implementing parallel fault simulation. In particular, we implement a pattern- and fault-parallel fault simulator. Our implementation fault-simulates a circuit in a levelized fashion. All threads of the GPU compute identical instructions, but on different data, as required by the single instruction multiple data (SIMD) programming semantics of the GPU. Fault injection is also done along with gate evaluation, with each thread using a different fault injection mask. Since GPUs have an extremely large memory bandwidth, we implement each of our fault simulation threads (which execute in parallel with no data dependencies) using memory lookup. Our experiments indicate that our approach, implemented on a single NVIDIA GeForce GTX 280 GPU card can simulate on average 47× faster when compared to the commercial fault simulation tool [1]. With the NVIDIA Tesla card, our approach would be potentially 300× faster.

References

1. Commercial fault simulation tool. Licensing agreement with the tool vendor requires that we do not disclose the name of the tool or its vendor.
2. IWLS 2005 Benchmarks. `http://www.iwls.org/iwls2005/benchmarks.html`
3. NVIDIA Tesla GPU Computing Processor. `http://www.nvidia.com/object/IO_43499.html`
4. Abramovici, A., Levendel, Y., Menon, P.: A logic simulation engine. In: IEEE Transactions on Computer-Aided Design, vol. 2, pp. 82–94 (1983)
5. Agrawal, P., Dally, W.J., Fischer, W.C., Jagadish, H.V., Krishnakumar, A.S., Tutundjian, R.: MARS: A multiprocessor-based programmable accelerator. IEEE Design and Test **4** (5), 28–36 (1987)
6. Amin, M.B., Vinnakota, B.: Workload distribution in fault simulation. Journal of Electronic Testing **10**(3), 277–282 (1997)
7. Amin, M.B., Vinnakota, B.: Data parallel fault simulation. IEEE Transactions on Very Large Scale Integration (VLSI) systems **7**(2), 183–190 (1999)
8. Banerjee, P.: Parallel Algorithms for VLSI Computer-aided Design. Prentice Hall Englewood Cliffs, NJ (1994)
9. Beece, D.K., Deibert, G., Papp, G., Villante, F.: The IBM engineering verification engine. In: DAC '88: Proceedings of the 25th ACM/IEEE Conference on Design Automation, pp. 218–224. IEEE Computer Society Press, Los Alamitos, CA (1988)
10. Gulati, K., Khatri, S.P.: Towards acceleration of fault simulation using graphics processing units. In: Proceedings, IEEE/ACM Design Automation Conference (DAC), pp. 822–827 (2008)

11. Ishiura, N., Ito, M., Yajima, S.: High-speed fault simulation using a vector processor. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD) (1987)
12. Mueller-Thuns, R., Saab, D., Damiano, R., Abraham, J.: VLSI logic and fault simulation on general-purpose parallel computers. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, pp. 446–460 (1993)
13. Narayanan, V., Pitchumani, V.: Fault simulation on massively parallel simd machines: Algorithms, implementations and results. Journal of Electronic Testing **3**(1), 79–92 (1992)
14. Ozguner, F., Aykanat, C., Khalid, O.: Logic fault simulation on a vector hypercube multiprocessor. In: Proceedings of the third conference on Hypercube concurrent computers and applications, pp. 1108–1116 (1988)
15. Ozguner, F., Daoud, R.: Vectorized fault simulation on the Cray X-MP supercomputer. In: Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers, IEEE International Conference on, pp. 198–201 (1988)
16. Parkes, S., Banerjee, P., Patel, J.: A parallel algorithm for fault simulation based on PROOFS. pp. 616–621. URL `citeseer.ist.psu.edu/article/parkes95parallel.html`
17. Patil, S., Banerjee, P.: Performance trade-offs in a parallel test generation/fault simulation environment. In: IEEE Transactions on Computer-Aided Design, pp. 1542–1558 (1991)
18. Pfister, G.F.: The Yorktown simulation engine: Introduction. In: DAC '82: Proceedings of the 19th Conference on Design Automation, pp. 51–54. IEEE Press, Piscataway, NJ (1982)
19. Raghavan, R., Hayes, J., Martin, W.: Logic simulation on vector processors. In: Computer-Aided Design, Digest of Technical Papers, IEEE International Conference on, pp. 268–271 (1988)
20. Tai, S., Bhattacharya, D.: Pipelined fault simulation on parallel machines using the circuitflow graph. In: Computer Design: VLSI in Computers and Processors, pp. 564–567 (1993)

Chapter 9
Fault Table Generation Using Graphics Processors

9.1 Chapter Overview

In this chapter, we explore the implementation of fault table generation on a graphics processing unit (GPU). A fault table is essential for fault diagnosis and fault detection in VLSI testing and debug. Generating a fault table requires extensive fault simulation, with no fault dropping, and is extremely expensive from a computational standpoint. Fault simulation is inherently parallelizable, and the large number of threads that a GPU can operate on in parallel can be employed to accelerate fault simulation, and thereby accelerate fault table generation. Our approach, called GFTABLE, employs a pattern-parallel approach which utilizes both bit parallelism and thread-level parallelism. Our implementation is a significantly modified version of FSIM, which is pattern-parallel fault simulation approach for single-core processors. Like FSIM, GFTABLE utilizes critical path tracing and the dominator concept to prune unnecessary simulations and thereby reduce runtime. Further modifications to FSIM allow us to maximally harness the GPU's huge memory bandwidth and high computational power. Our approach does not store the circuit (or any part of the circuit) on the GPU. Efficient parallel reduction operations are implemented in our implementation of GFTABLE. We compare our performance to FSIM$*$, which is FSIM modified to generate a fault table on a single-core processor. Our experiments indicate that GFTABLE, implemented on a single NVIDIA Quadro FX 5800 GPU card, can generate a fault table for 0.5 million test patterns, on average 15.68$\times$ faster when compared with FSIM$*$. With the NVIDIA Tesla server, our approach would be potentially 89.57$\times$ faster.

The remainder of this chapter is organized as follows. The motivation for this work is described in Section 9.2. Previous work in fault simulation and fault table generation has been described in Section 9.3. Section 9.4 details our approach for implementing fault simulation and table generation on GPUs. In Section 9.5 we present results of experiments which were conducted in order to benchmark our approach. We summarize the chapter in Section 9.6.

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_9,
© Springer Science+Business Media, LLC 2010

9.2 Introduction

With the increasing complexity and size of digital VLSI designs, the number of faulty variations of these designs is growing exponentially, thus increasing the time and effort required for *VLSI testing* and *debug*. Among the key steps in VLSI testing and debug are fault detection and diagnosis. *Fault detection* aims at differentiating a faulty design from a fault-free design, by applying test vectors. *Fault diagnosis* aims at identifying and isolating the fault, in order to analyze the defect causing the faulty behavior, with the help of test vectors which detect the fault. Both detection and diagnosis [4, 23, 24] require precomputed information about whether vector v_i can detect fault f_j, for all i and j. This information is stored in the form of a precomputed *fault table*. In general, a fault table is a matrix $[a_{ij}]$ where columns represent faults, rows represent test vectors, and $a_{ij} = 1$ if the test vector v_i detects the fault f_j, else $a_{ij} = 0$.

A fault table (also called a *pass/fail fault dictionary* [22]) is generated by extensive *fault simulation*. Given a digital design and a set of input vectors V defined over its primary inputs, fault simulation evaluates (for all i) the set of stuck-at faults F^i_{sim} that are tested by applying the vectors $v_i \in V$. The faults tested by each vector are then recorded in the matrix format of the fault table described earlier. Since the detectability of every fault is evaluated for every vector, the compute time for generating a fault table is extremely large. If a fault is *dropped* from the fault list as soon as a vector successfully detects it, the compute time can be reduced. However, the fault table thus generated may be insufficient for fault diagnosis. Thus, fault dropping cannot be performed during the generation of the fault table. For fault detection, we would like to find a minimal set of vectors which can maximally detect the faults. In order to compute this minimal set of vectors, the generation of a fault table with limited or no fault dropping is required. From this information, we could solve a unate covering problem to find the minimum set of vectors that detects all faults. For these reasons, fault table generation without fault dropping is usually performed. As a result, the high runtime of fault table generation becomes a key concern, making it important to explore ways to accelerate fault table generation. The ideal approach should be fast, scalable, *and* cost effective.

In order to reduce the compute time for generating the fault table, parallel implementations of fault simulation have been routinely used [9]. Fault simulation can be parallelized by a variety of techniques. These techniques include parallelizing the fault simulation algorithm (*algorithm-parallel techniques* [7, 3, 6]), partitioning the circuit into disjoint components and simulating them in parallel (*model-parallel* techniques [17, 25]), partitioning the fault set data and simulating faults in parallel (*data-parallel* techniques [20, 10, 18]), and a combination of one or more of these [16]. Data-parallel techniques can be further classified into *fault-parallel* methods, wherein different faults are simulated in parallel, and *pattern-parallel* approaches, wherein different input patterns (for the same fault) are simulated in parallel. Pattern-parallel approaches, as described in [15, 19], exploit the inherent bit parallelism of logic operations on computer words. In this chapter, we present a fault table generation approach that utilizes a *pattern-parallel* approach implemented on

graphics processing units (GPUs). Our notion of pattern parallelism includes *bit parallelism* obtained by performing logical operations on words and *thread- level parallelism* obtained by running several GPU threads concurrently.

Our approach for fault table generation is based on the fault simulation algorithm called FSIM [15]. FSIM was developed to run on a single-core CPU. However, since the target hardware in our case is a SIMD GPU machine, and the objective is to accelerate fault table generation, the FSIM algorithm is augmented and its implementation significantly modified to maximally harness the computational power and memory bandwidth available in the GPU. Fault simulation of a logic netlist effectively requires multiple logic simulations of the *true value (or fault-free)* simulations, and simulations with faults injected at various gates (typically primary inputs and reconvergent fanout branches as per the checkpoint fault injection model [11]). This is a natural match for the GPU's capabilities, since it exploits the extreme memory bandwidths of the GPU, as well as the presence of several SIMD processing elements on the GPU. Further, the computer words on the latest GPUs today allow 32- or even 64-bit operations. This facilitates the use of bit parallelism to further speed up fault simulation. For scalability reasons, our approach *does not store the circuit (or any part of the circuit) on the GPU*.

This work is the first, to the best of the authors' knowledge, to accelerate fault table generation on a GPU platform. The key contributions of this work are as follows:

- We exploit the match between pattern-parallel (bit-parallel and also thread-parallel) fault simulation with the capabilities of a GPU (a SIMD-based device) and harness the computational power of GPUs to accelerate fault table generation.
- The implementation satisfies the key requirements which ensure maximal speedup in a GPU. These are as follows:
  - The different threads which perform gate evaluations and fault injections are implemented such that the data dependency between threads is minimized.
  - All threads compute identical instructions, but on different data, which conforms to the SIMD architecture of the GPU.
  - Fast parallel reduction on the GPU is employed for computing the logical OR of thousands of words containing fault simulation data.
  - The amount of data transfer between the GPU and the host (CPU) is minimized. To achieve this, the large on-board memory on the recent GPUs is maximally exploited.
- In comparison to FSIM$*$ (i.e., FSIM [15] modified to generate the fault dictionary), our implementation is on average 15.68$\times$ faster, for 0.5 million patterns, over the ISCAS and ITC99 benchmarks.
- Further, even though our current implementation has been benchmarked on a single NVIDIA Quadro FX 5800 graphics card, the NVIDIA Tesla GPU Computing Processor [1] allows up to eight NVIDIA Tesla GPUs (on a 1U server). We

estimate that our implementation, using the NVIDIA Tesla server, can generate a fault dictionary on average 89.57× faster, when compared to FSIM∗.

Our fault dictionary computation algorithm is implemented in the Compute Unified Device Architecture (CUDA), which is an open-source programming and interfacing tool provided by NVIDIA corporation, for programming NVIDIA's GPU devices. The correctness of our GPU-based fault table generator, GFTABLE, has been verified by comparing its results with the results of FSIM∗ (which is run on the CPU). An extended abstract of this work can be found in [12].

9.3 Previous Work

Efficient fault simulation is a requirement for generating a fault dictionary. We discussed some previous work in accelerating fault simulation in Chapter 8. We devote the rest of this section to a brief discussion on FSIM [15], the algorithm that our approach is based upon.

The underlying algorithm for our GPU-based fault table generation engine is based on an approach for accelerating fault simulation called FSIM [15]. FSIM is a data-parallel approach that is implemented on a single-core microprocessor. The essential idea of FSIM is to simulate the circuit in a levelized manner from inputs to outputs and to prune off unnecessary gates as early as possible. This is done by employing critical path tracing [14, 5] and the dominator concept [8, 13], both of which reduce the amount of explicit fault simulation required. Some details of FSIM are explained in Section 9.4. We use a modification of FSIM (which we call FSIM∗) to generate the fault table and compare the performance of our GPU-based fault-table generator (GFTABLE) with that of FSIM∗. Since the target hardware in our case is a GPU, the original algorithm is redesigned and augmented to maximally exploit the computational power of the GPU.

The approach described in Chapter 8 accelerates fault simulation by employing a table lookup-based approach on the GPU. Chapter 8, in contrast to the current chapter, does not target a fault table computation, but only accelerates fault simulation.

An approach which generates compressed fault tables or dictionaries is described in [22]. This approach focuses on reducing the size of the fault table by using compaction [4, 26] or aliasing [21] techniques during fault table generation. Our approach, on the other hand, reduces the compute time for fault table generation by exploiting the immense parallelism available in the GPU, and is hence orthogonal to [22].

9.4 Our Approach

In order to maximally harness the high computational power of the GPU, our fault table generation approach is designed in a manner that is aware of the GPU's architectural, functional features and constraints. For instance, the programming

model of a GPU is the single instruction multiple data (SIMD) model, under which all threads must compute identical instructions, but on different data. GPUs allow extreme speedups if the different threads being evaluated have minimal data dependencies or global synchronization requirements. Our implementation honors these constraints and maximally avoids data or control dependencies between different threads. Further, even though the GPU's maximum bandwidth to/from the on-board memory has dramatically increased in recent GPUs (to $\sim$ 141.7 GB/s in the NVIDIA Quadro FX 5800), the GPU to host communication in our implementation is done using the PCIe 2.0 standard, with a data rate of $\sim$500 MB/s for 16 lanes. Therefore, our approach is implemented such that the communication between the host and the GPU is minimized.

In this section, we provide the details of our GFTABLE approach. As mentioned earlier, we modified FSIM [15] (which only performs fault simulation) to generate a complete fault table on a single-threaded CPU and refer to this version as FSIM$*$. The underlying algorithm for GFTABLE is a significantly re-engineered variant of FSIM$*$. We next present some preliminary information, followed by a description of FSIM$*$, along with the modifications we made to FSIM$*$ to realize GFTABLE, which capitalizes on the parallelism available in a GPU.

9.4.1 Definitions

We first define some of the key terms with the help of the example circuit shown in Fig. 9.1. A *stem* (or *fanout stem*) is defined as a *line* (or net) which fans out to more than one gate. All primary outputs of the circuit are defined as stems. For example in Fig. 9.1, the stems are k and p. If the fanout branches of each stem are cut off, this induces a partition of the circuit into *fanout-free regions* (FFRs). For example, in Fig. 9.1, we get two FFRs as shown by the dotted triangles. The output

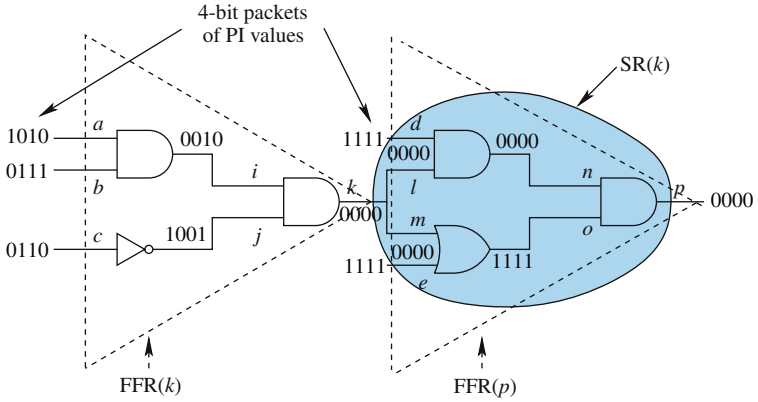


Fig. 9.1 Example circuit

of any FFR is a stem (say s), and the FFR is referred to as FFR(s). If all paths from a stem s pass through a line l before reaching a primary output, then the line l is called a *dominator* of the stem s. If there are no other dominators between the stem s and dominator l, then line l is called the *immediate dominator* of s. In the example, p is an immediate dominator of stem k in Fig. 9.1. The region between a stem s and its immediate dominator is called the *stem region* (SR) of s and is referred to as SR(s). Also, we define a *vector* as a two-dimensional array with a length equal to the number of primary inputs and a width equal to P, the *packet size*. In Fig. 9.1, the vectors are on the primary inputs a, b, c, d, and e. The packet size is $P = 4$. In other words, each vector consists of P fault patterns. In practice, the packet size for bit-parallel fault simulators is typically equal to the word size of the computer on which the simulator is implemented. In our experiments, the packet size (P) is 32.

If the change of the logic value at line s is observable at line t, then *detectability* $D(s, t) = 1$, else $D(s, t) = 0$. If a fault f injected at some line is detectable at line t, then *fault detectability* $FD(f, t) = 1$, else $FD(f, t) = 0$. If t is a primary output, the (fault) detectability is called a *global (fault) detectability*. The *cumulative detectability* of a line s, CD(s), is the logical OR of the fault detectabilities of the lines which merge at s. The ith element of CD(s) is defined as 1 iff there exists a fault f (to be simulated) such that $FD(f, s) = 1$ under the application of the ith test pattern of the vector. Otherwise, it is defined as 0. The following five properties hold for cumulative detectabilities:

- If a fault f (either s-a-1 or s-a-0) is injected at a line s and no other fault propagates to s, then $CD(s) = FD(f, s)$.
- If both s-a-0 and s-a-1 faults are injected at a line s, $CD(s) = (11 \ldots 1)$.
- If no fault is injected at a line s and no other faults propagate to s, then $CD(s) = (00\ldots0)$.
- Suppose there is a path from s to t. Then $CD(t) = CD(s) \cdot D(s, t)$, where $\cdot$ is the bitwise AND operation.
- Suppose two paths $r \rightarrow t$ and $s \rightarrow t$ merge. Then $CD(t) = (CD(r)D(r, t) + CD(s)D(s, t))$, where $+$ is the bitwise OR operation.

Further details on detectability and cumulative detectability can be found in [15].

The *sensitive inputs* of a unate gate with two or more inputs are determined as follows:

- If only one input k has a dominant logic value (*DLV*), then k is sensitive. AND and NAND gates have a *DLV* of 0. OR and NOR gates have a *DLV* of 1.
- If all the inputs of a gate have a value $\overline{DLV}$, then all inputs are sensitive.
- Otherwise no input is sensitive.

Critical path tracing (CPT), which was introduced in [3], is an alternative to conventional forward fault simulation. The approach consists of determining paths of *critical lines*, called critical paths, by a backtracing process starting at the POs for a vector v_i. Note that a *critical line* is a line driving the sensitive input of a gate. Note that the POs are critical in any test. By finding the critical lines for v_i, one

can immediately infer the faults detected by v_i. CPT is performed after fault-free simulation of the circuit for a vector v_i has been conducted. To aid the backtracing, sensitive gate inputs during fault-free simulation are marked.

For FFRs, CPT is always exact. In both approaches described in the next section, FSIM∗ and GPU-TABLE, CPT is used only for the FFRs. An example illustrating CPT is provided in the sequel.

9.4.2 Algorithms: FSIM∗ and GFTABLE

The algorithm for FSIM∗ is displayed in Algorithm 9. The key modifications for GFTABLE are explained in text in the sequel. Both FSIM∗ and GFTABLE maintain three major lists, a fault list (FL), a stem list (STEM_LIST), and an active stem list (ACTIVE_STEM), all on the CPU. The stem list stores all the stems $\{s\}$ whose corresponding FFRs ($\{FFR(s)\}$) are candidates for fault simulation. The active stem list stores stems $\{s^*\}$ for which at least one fault propagates to the immediate dominator of the stem s^*. The stems stored in the two lists are in the ascending order of their topological levels.

It is important to note that the GPU can never launch a kernel. Kernel launches are exclusively performed by the CPU (host). As a result, if (as in the case of GFTABLE) a conditional evaluation needs to be performed (lines 15, 17, and 25 for example), the condition *must* be checked by the CPU, which can then launch the appropriate GPU kernel if the condition is met. Therefore, the value being tested in the condition must be transferred by the GPU back to the CPU. The GPU operates on T threads at once (each computing a 32-bit result). Hence, in order to reduce the volume of data transferred and to reduce it to the size of a computer word on the CPU, the results from the GPU threads are reduced down to one 32-bit value before being transferred back to the CPU.

The argument to both the algorithms is the number of test patterns (N) over which the fault table is to be computed for the circuit. As a preprocessing step, both FSIM∗ and GFTABLE compute the fault list FL, award every gate a *gate_id*, compute the level of each gate, and identify the stems. The algorithms then identify the FFR and SR of each stem (this is computed on the CPU). As discussed earlier, the stems and the corresponding FFRs and SRs of these stems in our example circuit are marked in Fig. 9.1. Let us consider the following five faults in our example circuit: a s-a-0, c s-a-1, c s-a-0, l s-a-0, and l s-a-1, which are added to the fault list FL. Also assume that the fault table generation is carried out for a single vector of length 5 (since there are 5 primary inputs) consisting of 4-bit-wide packets. In other words, each vector consists of four patterns of primary input values. The fault table $[a_{ij}]$ is initialized to the all zero matrix. In our example, the size of this matrix is $N \times 5$. The above steps are shown in lines 1 through 5 of Algorithm 9. The rest of FSIM∗ and GFTABLE are within a while loop (line 7) with condition $v < N$, where N is the total number of patterns to be simulated and v is the current count of patterns which are already simulated. For both algorithms, v is initialized to zero (line 6).

Algorithm 9 Pseudocode of FSIM∗

```
1: FSIM∗(N){
2: Set up Fault list FL.
3: Find FFRs and SRs.
4: STEM_LIST ← all stems
5: Fault table [a_ik] initialized to all zero matrix.
6: v=0
7: while v < N do
8:    v=v + packet width
9:    Generate one test vector using LFSR
10:   Perform fault free simulation
11:   ACTIVE_STEM ← NULL.
12:   for each stem s in STEM_LIST do
13:      Simulate FFR using CPT
14:      Compute CD(s)
15:      if (CD(s) ≠ (00...0)) then
16:         Simulate SRs and compute D(s, t), where t is the immediate dominator of s.
17:         if (D(s, t) ≠ (00...0)) then
18:            ACTIVE_STEM ← ACTIVE_STEM + s.
19:         end if
20:      end if
21:   end for
22:   while (ACTIVE_STEM ≠ NULL) do
23:      Remove the highest level stem s from ACTIVE_STEM.
24:      Compute D(s, t), where t is an auxiliary output which connects all primary outputs.
25:      if (D(s, t) ≠ (00...0)) then
26:         for (each fault f_i in FFR(s)) do
27:            FD(f_i, t) = FD(f_i, s) · D(s, t).
28:            Store FD(f_i, t) in the ith row of [a_ik]
29:         end for
30:      end if
31:   end while
32: end while
33: }
```

9.4.2.1 Generating Vectors (Line 9)

The test vectors in FSIM∗ are generated using an LFSR-based pseudo-random number generator on the CPU. For every test vector, as will be seen later, fault-free and faulty simulations are carried out. Each test vector in FSIM∗ is a vector (array) of 32-bit integers with a length equal to the number of primary inputs (NPI). In this case, v is incremented by 32 (packet width) in every iteration of the while loop (line 8).

Each test vector in GFTABLE is a vector of length NPI and width $32 \times T$, where T is the number of threads launched in parallel in a grid of thread blocks. Therefore, in this case, for every while loop iteration, v is incremented by $T \times 32$. The test vectors are generated on the CPU (as in FSIM∗) and transferred to the GPU memory. In all the results reported in this chapter, both FSIM∗ and GFTABLE utilize identical

test vectors (generated by the LFSR-based pseudo-random number generator on the CPU). In all examples, the results of GFTABLE matched those of FSIM*. The GFTABLE runtimes reported always include the time required to transfer the input patterns to the GPU and the time required to transfer results back to the CPU.

9.4.2.2 Fault-Free Simulation (Line 10)

Now, for each test vector, FSIM∗ performs fault-free or true value simulation. Fault-free simulation is essentially the logic simulation of every gate, carried out in a forward levelized order. The fault-free output at every gate, computed as a result of the gate's evaluation, is recorded in the CPU's memory.

Fault-free simulation in GFTABLE is carried out in a forward levelized manner as well. Depending on the gate type and the number of inputs, a separate kernel on the GPU is launched for T threads. As an example, the pseudocode of the kernel which evaluates the fault-free simulation value of a 2-input AND gate is provided in Algorithm 10. The arguments to the kernel are the pointer to global memory, *MEM*, where fault-free values are stored, and the gate_id of the gate being evaluated (*id*) and its two inputs (a and b). Let the thread's (unique) *threadID* be t_x. The data in *MEM*, indexed at a location ($t_x + a \times T$), is ANDed with the data at location ($t_x + b \times T$) and the result is stored in *MEM* indexed at location ($t_x + id \times T$). Our implementation has a similar kernel for every gate in our library.

Since the amount of global memory on the GPU is limited, we store the fault-free simulation data in the global memory of the GPU for at most L gates[1] of a circuit. Note that we require two copies of the fault-free simulation data, one for use as a reference and the other for temporary modification to compute faulty-circuit data. For the gates whose fault-free data is not stored on the GPU, the fault-free data is transferred to and from the CPU, as and when it is computed or required on the GPU. This allows our GFTABLE approach to scale *regardless of the size of the given circuit*.

Figure 9.1 shows the fault-free output at every gate, when a single test vector of packet width 4 is applied at its 5 inputs.

Algorithm 10 Pseudocode of the Kernel for Logic Simulation of 2-Input AND Gate

```
logic_simulation_kernel_AND_2(int *MEM, int id, int a, int b){
t_x = my_thread_id
MEM[t_x + id * T] = MEM[t_x + a * T] · MEM[t_x + b * T]
}
```

[1] We store fault-free data for the L gates of the circuit that are topologically closest to the primary inputs of the circuit.

9.4.2.3 Computing Detectabilities and Cumulative Detectabilities (Lines 13, 14)

Next, in the FSIM∗ and GFTABLE algorithms, for every stem s, CD(s) is computed. This is done by computing the detectability of every fault in FFR(s) by using critical path tracing and the properties of cumulative detectabilities discussed in Section 9.4.1.

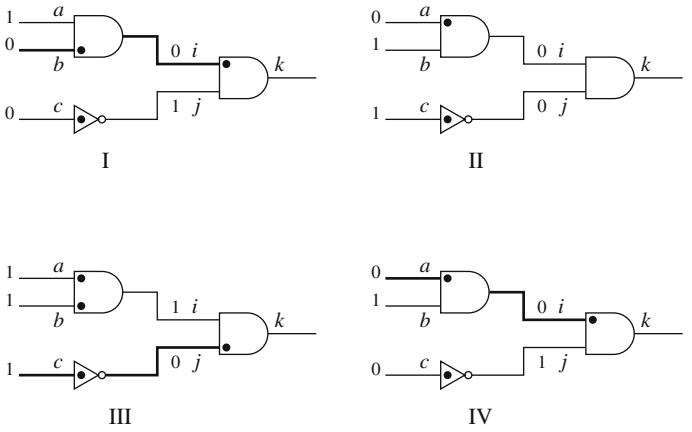


Fig. 9.2 CPT on FFR(k)

This step is further explained by the help of Fig. 9.2. The FFR(k) from the example circuit is copied four times,[2] one for each pattern in the vector applied. In each of the copies, the sensitive input is marked using a bold dot. The critical lines are darkened. Using these markings, the detectabilities of all lines at stem k can be computed as follows: D(a, k) = 0001. This is because out of the four copies, only in the fourth copy a lies on the sensitive path (i.e., a path consisting of critical lines) backtraced from k. Similarly we compute the following:
D(b, k) = 1000; D(c, k) = 0010; D(i, k) = 1001; D(j, k)=0010; D(k, k) = 1111; D(a, i) = 0111; D(b, i) = 1010; and D(c, j) = 1111

Now for the faults in FFR(k) (i.e., a s-a-0, c s-a-0, and c s-a-1), we compute the FDs as follows:
FD(a s-a-0, k) = FD(a s-a-0, a) · D(a, k)
For every test pattern, the fault a s-a-0 can be observed at a only when the fault-free value at a is different from the stuck-at value of the fault. Among the four copies in Fig. 9.2, only the first and third copies have a fault-free value of '1' at line a, and thus fault a s-a-0 can be observed only in the first and third copies. Therefore FD(a s-a-0, a) = 1010. Therefore, FD(a s-a-0, k) = 1010 · 0001 = 0000. Similarly, FD(c s-a-0, k) = 0010 and FD(c s-a-1, k) = 0000.

[2] This is because the packet width is 4.

Now, by definition
CD(k) = (CD(i) · D(i, k) + CD(j) · D(j, k)) and CD(i) = (CD(a) · D(a, i) + CD(b) · D(b, i))
From the first property discussed for CD, CD(a) = FD(a s-a-0, a) = 1010, and by definition CD(b) = 0000. By substitution and similarly computing CD(i) and CD(j), we compute CD(k) = 0010.

The implementation of the computation of detectabilities and cumulative detectabilities in FSIM∗ and GFTABLE is different, since in GFTABLE, all computations for computing detectabilities and cumulative detectabilities are done on the GPU, with every kernel executed on the GPU launched with T threads. Thus a single kernel in GFTABLE computes T times more data, compared to the corresponding computation in FSIM∗. In FSIM∗, the backtracing is performed in a topological manner from the output of the FFR to its inputs and is not scheduled for gates driving zero critical lines in the packet. We found that this pruning reduces the number of gate evaluations by 42% in FSIM∗ (based on tests run on four benchmark circuits). In GFTABLE, however, T times more patterns are evaluated at once, and as a result, no reduction in the number of scheduled gate evaluations were observed for the same four benchmarks. Hence, in GFTABLE, we perform a brute-force backtracing on *all* gates in an FFR.

As an example, the pseudocode of the kernel which evaluates the cumulative detectability at output k of a 2-input gate with inputs i and j is provided in Algorithm 11. The arguments to the kernel are the pointer to global memory, *CD*, where cumulative detectabilities are stored; pointer to global memory, *D*, where detectabilities to the immediate dominator are stored; the gate_id of the gate being evaluated (k) and its two inputs (i and j). Let the thread's (unique) *threadID* be t_x. The data in *CD* and *D*, indexed at a location ($t_x + i \times T$) and ($t_x + j \times T$), and the result computed as per
CD(k) = (CD(i) · D(i, k) + CD(j) · D(j, k))
is stored in *CD* indexed at location ($t_x + k \times T$). Our implementation has a similar kernel for 2-, 3-, and 4-input gates in our library.

Algorithm 11 Pseudocode of the Kernel to Compute CD of the Output k of 2-Input Gate with Inputs i and j

```
CPT_kernel_2(int * CD,int * D,inti,intj,intk){
t_x = my_thread_id
CD[t_x + k * T] = CD[t_x + i * T] · D[t_x + i * T] + CD[t_x + j * T] · D[t_x + j * T]
}
```

9.4.2.4 Fault Simulation of SR(s) (Lines 15, 16)

In the next step, the FSIM∗ algorithm checks that CD(s) $\neq$ (00...0) (line 15), before it schedules the simulation of SR(s) until its immediate dominator t and the computation of D(s, t). In other words, if CD(s) = (00...0), it implies that for the current vector, the frontier of all faults upstream from s has died before reaching the stem

s, and thus no fault can be detected at s. In that case, the fault simulation of SR(s) would be pointless.

In the case of GFTABLE, the effective packet size is $32 \times T$. T is usually set to more than 1,000 (in our experiments it is $\geq$10K), in order to take advantage of the parallelism available on the GPU and to amortize the overhead of launching a kernel and accessing global memory. The probability of finding CD(s) = (00...0) in GFTABLE is therefore very low ($\sim$0.001). Further, this check would require the logical OR of T 32-bit integers on the GPU, which is an expensive computation. As a result, we bypass the test of line 15 in GFTABLE and always schedule the computation of SR(s) (line 16).

In simulating SR(s), explicit fault simulation is performed in the forward levelized order from stem s to its immediate dominator t. The input at stem s during simulation of SR(s) is CD(s) XORed with fault-free value at s. This is equivalent to injecting the faults which are upstream from s and observable at s. After the fault simulation of SR(s), the detectability D(s, t) is computed by XORing the simulation output at t with the true value simulation at t. During the forward levelized simulation, the immediate fanout of a gate g is scheduled only if the result of the logic evaluation at g is different from its fault-free value. This check is conducted for every gate in all paths from stem s to its immediate dominator t. On the GPU, this step involves XORing the current gate's T 32-bit outputs with the previously stored fault-free T 32-bit outputs. It would then require the computation of a logical reduction OR of the T 32-bit results of the XOR into one 32-bit result. This is because line 17 is computed on the CPU, which requires a 32-bit operand. In GFTABLE, the reduction OR operation is a modified version of the highly optimized tree-based parallel reduction algorithm on the GPU, described in [2]. The approach in [2] effectively avoids bank conflicts and divergent warps, minimizes global memory access latencies, and employs loop unrolling to gain further speedup. Our modified reduction algorithm has a key difference compared to [2]. The approach in [2] computes a SUM instead of a logical OR. The approach described in [2] is a breadth-first approach. In our case, employing a breadth-first approach is expensive, since we need to detect if *any* of the $T \times 32$ bits is not equal to 0. Therefore, as soon as we find a single non-zero entry we can finish our computation. Note that performing this test sequentially would be extremely slow in the worst case. We therefore equally divide the array of T 32-bit words into smaller groups of size Q words and compute the logical OR of all numbers within a group using our modified parallel reduction approach. As a result, our approach is a hybrid of a breadth-first and a depth-first approach. If the reduction result for any group is not (00...0), we return from the parallel reduction kernel and schedule the fanout of the current gate. If the reduction result for any group, on the other hand, is equal to (00...0), we compute the logical reduction OR of the next group and so on. Each logical reduction OR is computed using our reduction kernel, which takes advantage of all the optimizations suggested in [2] (and improves [2] further by virtue of our modifications). The optimal size of the reduction groups was experimentally determined to be $Q = 256$. We found that when reducing 256 words at once, there was a high probability of having at least one non-zero bit, and thus there was a high likelihood of returning early from the parallel

reduction kernel. At the same time, using 256 words allowed for a fast reduction within a single thread block of size equal to 128 threads. Scheduling a thread block of 128 threads uses 4 warps (of warp size equal to 32 threads each). The thread block can schedule the 4 warps in a time-sliced fashion, where each integer OR operation takes 4 clock cycles, thereby making optimal use of the hardware resources.

Despite using the above optimization in parallel reduction, the check can still be expensive, since our parallel reduction kernel is launched after every gate evaluation. To further reduce the runtime, we launch our parallel reduction kernel after every G gate evaluations. During in-between runs, the fanout gates are always scheduled to be evaluated. Due to this, we would potentially do a few extra simulations, but this approach proved to be significantly faster when compared to either performing a parallel reduction after every gate's simulation or scheduling every gate in SR(s) for simulation in a brute-force manner. We experimentally determined the optimal value for G to be 20.

In the next step (lines 17 and 18), the detectability D(s, t) is tested. If it is not equal to (00...0), stem s is added to the ACTIVE_STEM list. Again this step of the algorithm is identical for FSIM∗ and GFTABLE; however, the difference is in the implementation. On the GPU, a parallel reduction technique (as explained above) is used for testing if D(s, t) $\neq$ (00...0). The resulting 32-bit value is transferred back to the CPU. The *if* condition (line 17) is checked on the CPU and if it is true, the ACTIVE_STEM list is augmented on the CPU.

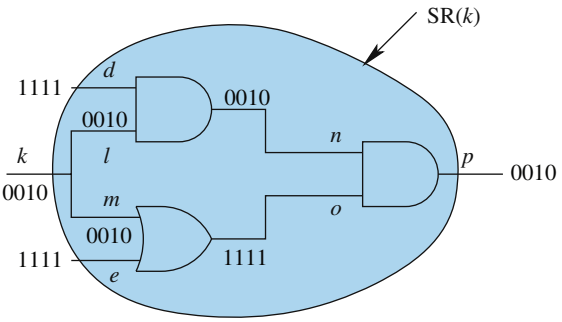


Fig. 9.3 Fault simulation on SR(k)

For our example circuit, SR(k) is displayed in Fig. 9.3. The input at stem k is 0010 (CD(k) XORed with fault-free value at k). The two primary inputs d and e have the original test vectors. From the output evaluated after explicit simulation until p, D(k,p) = 0010 $\neq$ 0000. Thus, k is added to the active stem list.

CPT on FFR(p) can be computed in a similar manner. The resulting values are listed below:
D(l, p)=1111; D(n, p)=1111; D(d, p)=0000; D(m, p)=0000; D(e, p)=0000; D(o,p) =0000; D(d, n)=0000; D(l, n)=1111; D(m, o)=0000; D(e, o)=1111; FD(l s-a-0, p)=0000; FD(l s-a-1, p)=1111; CD(d) = 0000; CD(l)=1111; CD(m)=0000; CD(e) =0000; CD(n)=1111; CD(o)=0000; and CD(p)=1111.

Since CD(p) $\neq$ (0000) and D(p, p) $\neq$ (0000), the stem p is added to ACTIVE_STEM list.

9.4.2.5 Generating the Fault Table (Lines 22–31)

Next, FSIM$*$ computes the global detectability of faults (and stems) in the backward order, i.e., it removes the highest level stem s from the ACTIVE_STEM list (line 23) and computes its global detectability (line 24). If it is not equal to (00...0) (line 25), the global detectability of every fault in FFR(s) is computed and stored in the $[a_{ij}]$ matrix (lines 26–28).

The corresponding implementation in GFTABLE maintains the ACTIVE_STEM on the CPU and, like FSIM$*$, first computes the global detectability of the highest level stem s from ACTIVE_STEM list, but on the GPU. Also, another parallel reduction kernel is invoked for D(s, t), since the resulting data needs to be transferred to the CPU for testing whether the global detectability of s is not equal to (00...0) (line 25). If true, the global detectability of every fault in FFR(s) is computed on the GPU and transferred back to the CPU to store the final fault table matrix on the CPU.

The complete algorithm of our GFTABLE approach is displayed in Algorithm 12.

9.5 Experimental Results

As discussed previously, pattern parallelism in GFTABLE includes both *bit-parallelism*, obtained by performing logical operations on words (i.e., packet size is 32), and *thread-level parallelism*, obtained by launching T GPU threads concurrently. With respect to bit parallelism, the bit width used in GFTABLE implemented on the NVIDIA Quadro FX 5800 was 32. This was chosen to make a fair comparison with FSIM$*$, which was run on a 32-bit, 3.6 GHz Intel CPU running Linux (Fedora Core 3), with 3 GB RAM. It should be noted that Quadro FX 5800 also allows operations on 64-bit words.

With respect to thread-level parallelism, launching a kernel with a higher number of threads in the grid allows us to better take advantage of the immense parallelism available on the GPU, reduces the overhead of launching a kernel, and hides the latency of accessing global memory. However, due to a finite size of the global memory there is an upper limit on the number of threads that can be launched simultaneously. Hence we split the fault list of a circuit into smaller fault lists. This is done by first sorting the gates of the circuit in increasing order of their level. We then collect the faults associated with every Z (=100) gates from this list, to generate the smaller fault lists. Our approach is then implemented such that a new fault list is targeted in a new iteration. We statically allocate global memory for storing the fault detectabilities of the current faults (faults currently under consideration) for all threads launched in parallel on the GPU. Let the number of faults in the current list being considered be F, and the number of threads launched simultaneously be T, then $F \times T \times 4\,\mathrm{B}$ of global memory is used for storing the current fault

Algorithm 12 Pseudocode of GFTABLE

```
GFTABLE(N){
Set up Fault list FL.
Find FFRs and SRs.
STEM_LIST ← all stems
Fault table [a_ik] initialized to all zero matrix.
v=0
while v < N do
  v=v + T × 32
  Generate using LFSR on CPU and transfer test vector to GPU
  Perform fault free simulation on GPU
  ACTIVE_STEM ← NULL.
  for each stem s in STEM_LIST do
    Simulate FFR using CPT on GPU // bruteforce backtracking on all gates
    Simulate SRs on GPU
    // check at every Gth gate during
    // forward levelized simulation if fault frontier still alive,
    // else continue with for loop with s ← next stem in STEM_LIST
    Compute D(s, t) on GPU, where t is the immediate dominator of s. // computed using
    hybrid parallel reduction on GPU
    if (D(s, t) ≠ (00...0)) then
      update on CPU ACTIVE_STEM ← ACTIVE_STEM + s
    end if
  end for
  while (ACTIVE_STEM ≠ NULL) do
    Remove the highest level stem s from ACTIVE_STEM.
    Compute D(s, t) on GPU, where t is an auxiliary output which connects all primary out-
    puts. // computed using hybrid parallel reduction on GPU
    if (D(s, t) ≠ (00...0)) then
      for (each fault f_i in FFR(s)) do
        FD(f_i, t) = FD(f_i, s) · D(s, t). // computed on GPU
        Store FD(f_i, t) in the ith row of [a_ik] // stored on CPU
      end for
    end if
  end while
end while
}
```

detectabilities. As mentioned previously, we statically allocate space for two copies of fault-free simulation output for at most L gates. The gates of the circuit are topologically sorted from the primary outputs to the primary inputs. The fault-free data (and its copy) of the first L gates in the sorted list are statically stored on the GPU. This further uses $L \times T \times 2 \times 4\,\mathrm{B}$ of global memory. For the remaining gates, the fault-free data is transferred to and from the CPU as and when it is computed or required on the GPU.

Further, the detectabilities and cumulative detectabilities of all gates in the FFRs of the current faults, and for all the dominators in the circuit, are stored on the GPU. The total on-board memory on a single NVIDIA Quadro FX 5800 is 4 GB. With our current implementation, we can launch T = 16K threads in

Table 9.1 Fault table generation results with $L = 32K$

Circuit	# Gates	# Faults	GFTABLE	FSIM∗	Speedup	GFTABLE-8	Speedup
c432	196	524	0.77	12.60	16.43×	0.13	93.87×
c499	243	758	0.75	8.40	11.20×	0.13	64.00×
c880	443	942	1.15	17.40	15.13×	0.20	86.46×
c1355	587	1,574	2.53	23.95	9.46×	0.44	54.03×
c1908	913	1,879	4.68	51.38	10.97×	0.82	62.70×
c2670	1,426	2,747	1.92	56.27	29.35×	0.34	167.72×
c3540	1,719	3,428	7.55	168.07	22.26×	1.32	127.20×
c5315	2,485	5,350	4.50	109.05	24.23×	0.79	138.48×
c6288	2,448	7,744	28.28	669.02	23.65×	4.95	135.17×
c7552	3,719	7,550	10.70	204.33	19.10×	1.87	109.12×
b14_1	7,283	12,608	70.27	831.27	11.83×	12.30	67.60×
b14	9,382	16,207	100.87	1,502.47	14.90×	17.65	85.12×
b15	12,587	21,453	136.78	1,659.10	12.13×	23.94	69.31×
b20_1	17,157	31,034	193.72	3,307.08	17.07×	33.90	97.55×
b20	20,630	35,937	319.82	4,992.73	15.61×	55.97	89.21×
b21_1	16,623	29,119	176.75	3,138.08	17.75×	30.93	101.45×
b21	20,842	35,968	262.75	4,857.90	18.49×	45.98	105.65×
b17	40,122	69,111	903.22	4,921.60	5.45×	158.06	31.14×
b18	40,122	69,111	899.32	4,914.93	5.47×	157.38	31.23×
b22_1	25,011	44,778	369.34	4,756.53	12.88×	64.63	73.59×
b22	29,116	51,220	399.34	6,319.47	15.82×	69.88	90.43×
Average					15.68×		89.57×

parallel, while using $L = 32K$ gates. Note that the complete fault dictionary is never stored on the GPU, and hence the number of test patterns used for generating the fault table can be arbitrarily large. Also, since GFTABLE does not store the information of the entire circuit on the GPU, it can handle arbitrary-sized circuits.

The results of our current implementation, for 10 ISCAS benchmarks and 11 ITC99 benchmarks, for 0.5M patterns, are reported in Table 9.1. All runtimes reported are in seconds. The fault tables obtained from GFTABLE, for all benchmarks, were verified against those obtained from FSIM∗ and were found to verify with 100% fidelity. Column 1 lists the circuit under consideration; columns 2 and 3 list the number of gates and (collapsed) faults in the circuit. The total runtimes for GFTABLE and FSIM∗ are listed in columns 4 and 5, respectively. The runtime of GFTABLE includes the *total* time taken on both the GPU and the CPU and the time taken for *all* the data transfers between the GPU and the CPU. In particular, the transfer time includes the time taken to transfer the following:

- the test patterns which are generated on the CPU (CPU → GPU);
- the results from the multiple invocations of the parallel reduction kernel (GPU → CPU);
- the global fault detectabilities over all test patterns for all faults (GPU → CPU); and

- the fault-free data of any gate which is not in the set of L gates (during true value and faulty simulations) (CPU $\leftrightarrow$ GPU).

Column 6 reports the speedup of GFTABLE over FSIM$*$. The average speedup over the 21 benchmarks is reported in the last row. On average, GFTABLE is 15.68$\times$ faster than FSIM$*$.

By using the NVIDIA Tesla server housing up to eight GPUs [1], the available global memory increases by 8$\times$. Hence we can potentially launch 8$\times$ more threads simultaneously and set L to be large enough to hold the fault-free data (and its copy) for all the gates in our benchmark circuits. This allows for a $\sim$8$\times$ speedup in the processing time. The first three items of the transfer times in the list above will not scale, and the last item will not contribute to the total runtime. In Table 9.1, column 7 lists the projected runtimes when using a 8 GPU system for GFTABLE (referred to as GFTABLE-8). The projected speedup of GFTABLE-8 compared to FSIM$*$ is listed in column 8. The average potential speedup is 89.57$\times$.

Tables 9.2 and 9.3 report the results with L = 8K and 16K, respectively. All columns in Tables 9.2 and 9.3 report similar entries as described for Table 9.1. The speedup of GFTABLE and GFTABLE-8 over FSIM$*$ with L = 8K is 12.88$\times$ and 69.73$\times$, respectively. Similarly, the speedup of GFTABLE and GFTABLE-8 over FSIM$*$ with L = 16K is 14.49$\times$ and 82.80$\times$, respectively.

Table 9.2 Fault table generation results with L = 8K

Circuit	# Gates	# Faults	GFTABLE	FSIM$*$	Speedup	GFTABLE-8	Speedup
c432	196	524	0.73	12.60	17.19$\times$	0.13	98.23$\times$
c499	243	758	0.75	8.40	11.20$\times$	0.13	64.00$\times$
c880	443	942	1.13	17.40	15.36$\times$	0.20	87.76$\times$
c1355	587	1,574	2.52	23.95	9.52$\times$	0.44	54.37$\times$
c1908	913	1,879	4.73	51.38	10.86$\times$	0.83	62.04$\times$
c2670	1,426	2,747	1.93	56.27	29.11$\times$	0.34	166.34$\times$
c3540	1,719	3,428	7.57	168.07	22.21$\times$	1.32	126.92$\times$
c5315	2,485	5,350	4.53	109.05	24.06$\times$	0.79	137.47$\times$
c6288	2,448	7,744	28.17	669.02	23.75$\times$	4.93	135.72$\times$
c7552	3,719	7,550	10.60	204.33	19.28$\times$	1.85	110.15$\times$
b14_1	7,283	12,608	70.05	831.27	11.87$\times$	12.26	67.81$\times$
b14	9,382	16,207	120.53	1,502.47	12.47$\times$	21.09	71.23$\times$
b15	12,587	21,453	216.12	1,659.10	7.68$\times$	37.82	43.87$\times$
b20_1	17,157	31,034	410.68	3,307.08	8.05$\times$	71.87	46.02$\times$
b20	20,630	35,937	948.06	4,992.73	5.27$\times$	165.91	30.09$\times$
b21_1	16,623	29,119	774.45	3,138.08	4.05$\times$	135.53	23.15$\times$
b21	20,842	35,968	974.03	4,857.90	5.05$\times$	170.46	28.50$\times$
b17	40,122	69,111	1,764.01	4,921.60	2.79$\times$	308.70	15.94$\times$
b18	40,122	69,111	2,100.40	4,914.93	2.34$\times$	367.57	13.37$\times$
b22_1	25,011	44,778	647.15	4,756.53	7.35$\times$	113.25	42.00$\times$
b22	29,116	51,220	915.87	6,319.47	6.90$\times$	160.28	39.43$\times$
Average					12.88$\times$		69.73$\times$

Table 9.3 Fault table generation results with $L = 16K$

Circuit	# Gates	# Faults	GFTABLE	FSIM∗	Speedup	GFTABLE-8	Speedup
c432	196	524	0.73	12.60	17.33×	0.13	99.04×
c499	243	758	0.75	8.40	11.20×	0.13	64.00×
c880	443	942	1.03	17.40	16.89×	0.18	96.53×
c1355	587	1,574	2.53	23.95	9.46×	0.44	54.03×
c1908	913	1,879	4.68	51.38	10.97×	0.82	62.70×
c2670	1,426	2,747	1.97	56.27	28.61×	0.34	163.46×
c3540	1,719	3,428	7.92	168.07	21.22×	1.39	121.26×
c5315	2,485	5,350	4.50	109.05	24.23×	0.79	138.48×
c6288	2,448	7,744	28.28	669.02	23.65×	4.95	135.17×
c7552	3,719	7,550	10.70	204.33	19.10×	1.87	109.12×
b14_1	7,283	12,608	70.27	831.27	11.83×	12.30	67.60×
b14	9,382	16,207	100.87	1,502.47	14.90×	17.65	85.12×
b15	12,587	21,453	136.78	1,659.10	12.13×	23.94	69.31×
b20_1	17,157	31,034	193.72	3,307.08	17.07×	33.90	97.55×
b20	20,630	35,937	459.82	4,992.73	10.86×	80.47	62.05×
b21_1	16,623	29,119	156.75	3,138.08	20.02×	27.43	114.40×
b21	20,842	35,968	462.75	4,857.90	10.50×	80.98	59.99×
b17	40,122	69,111	1,203.22	4,921.60	4.09×	210.56	23.37×
b18	40,122	69,111	1,399.32	4,914.93	3.51×	244.88	20.07×
b22_1	25,011	44,778	561.34	4,756.53	8.47×	98.23	48.42×
b22	29,116	51,220	767.34	6,319.47	8.24×	134.28	47.06×
Average					14.49×		82.80×

9.6 Chapter Summary

In this chapter, we have presented our implementation of fault table generation on a GPU, called GFTABLE. Fault table generation requires fault simulation without fault dropping, which can be extremely computationally expensive. Fault simulation is inherently parallelizable, and the large number of threads that can be computed in parallel on a GPU can therefore be employed to accelerate fault simulation and fault table generation. In particular, we implemented a pattern-parallel approach which utilizes both bit parallelism and thread-level parallelism. Our implementation is a significantly re-engineered version of FSIM, which is a pattern-parallel fault simulation approach for single-core processors. At no time in the execution is the entire circuit (or a part of the circuit) required to be stored (or transferred) on (to) the GPU. Like FSIM, GFTABLE utilizes critical path tracing and the dominator concept to reduce explicit simulation time. Further modifications to FSIM allow us to maximally harness the GPU's computational resources and large memory bandwidth. We compared our performance to FSIM∗, which is FSIM modified to generate a fault table. Our experiments indicate that GFTABLE, implemented on a single NVIDIA Quadro FX 5800 GPU card, can generate a fault table for 0.5 million test patterns, on average 15× faster when compared with FSIM∗. With the NVIDIA Tesla server [1], our approach would be potentially 90× faster.

References

1. NVIDIA Tesla GPU Computing Processor. http://www.nvidia.com/object/IO_43499.html
2. Parallel Reduction. http://developer.download.nvidia.com/~reduction.pdf
3. Abramovici, A., Levendel, Y., Menon, P.: A logic simulation engine. In: IEEE Transactions on Computer-Aided Design, vol. 2, pp. 82–94 (1983)
4. Abramovici, M., Breuer, M.A., Friedman, A.D.: Digital Systems Testing and Testable Design. Computer Science Press, New York (1990)
5. Abramovici, M., Menon, P.R., Miller, D.T.: Critical path tracing – An alternative to fault simulation. In: DAC '83: Proceedings of the 20th Conference on Design Automation, pp. 214–220. IEEE Press, Piscataway, NJ (1983)
6. Agrawal, P., Dally, W.J., Fischer, W.C., Jagadish, H.V., Krishnakumar, A.S., Tutundjian, R.: MARS: A multiprocessor-based programmable accelerator. IEEE Design and Test **4**(5), 28–36 (1987)
7. Amin, M.B., Vinnakota, B.: Workload distribution in fault simulation. Journal of Electronic Testing **10**(3), 277–282 (1997)
8. Antreich, K., Schulz, M.: Accelerated fault simulation and fault grading in combinational circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **6**(5), 704–712 (1987)
9. Banerjee, P.: Parallel Algorithms for VLSI Computer-aided Design. Prentice Hall Englewood Cliffs, NJ (1994)
10. Beece, D.K., Deibert, G., Papp, G., Villante, F.: The IBM engineering verification engine. In: DAC '88: Proceedings of the 25th ACM/IEEE Conference on Design Automation, pp. 218–224. IEEE Computer Society Press, Los Alamitos, CA (1988)
11. Bossen, D.C., Hong, S.J.: Cause-effect analysis for multiple fault detection in combinational networks. IEEE Transactions on Computers **20**(11), 1252–1257 (1971)
12. Gulati, K., Khatri, S.P.: Fault table generation using graphics processing units. In: IEEE International High Level Design Validation and Test Workshop (2009)
13. Harel, D., Sheng, R., Udell, J.: Efficient single fault propagation in combinational circuits. In: Proceedings of the International Conference on Computer-Aided Design ICCAD, pp. 2–5 (1987)
14. Hong, S.J.: Fault simulation strategy for combinational logic networks. In: Proceedings of Eighth International Symposium on Fault-Tolerant Computing, pp. 96–99 (1979)
15. Lee, H.K., Ha, D.S.: An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation. In: Proceedings of the IEEE International Test Conference on Test, pp. 946–955. IEEE Computer Society, Washington, DC (1991)
16. Mueller-Thuns, R., Saab, D., Damiano, R., Abraham, J.: VLSI logic and fault simulation on general-purpose parallel computers. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, pp. 446–460 (1993)
17. Narayanan, V., Pitchumani, V.: Fault simulation on massively parallel simd machines: Algorithms, implementations and results. Journal of Electronic Testing **3**(1), 79–92 (1992)
18. Ozguner, F., Daoud, R.: Vectorized fault simulation on the Cray X-MP supercomputer. In: Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers, IEEE International Conference on, pp. 198–201 (1988)
19. Parkes, S., Banerjee, P., Patel, J.: A parallel algorithm for fault simulation based on PROOFS. pp. 616–621.
    URL citeseer.ist.psu.edu/article/parkes95parallel.html
20. Pfister, G.F.: The Yorktown simulation engine: Introduction. In: DAC '82: Proceedings of the 19th Conference on Design Automation, pp. 51–54. IEEE Press, Piscataway, NJ (1982)
21. Pomeranz, I., Reddy, S., Tangirala, R.: On achieving zero aliasing for modeled faults. In: Proc. [3rd] European Conference on Design Automation, pp. 291–299 (1992)

22. Pomeranz, I., Reddy, S.M.: On the generation of small dictionaries for fault location. In: ICCAD '92: 1992 IEEE/ACM International Conference Proceedings on Computer-Aided Design, pp. 272–279. IEEE Computer Society Press, Los Alamitos, CA (1992)
23. Pomeranz, I., Reddy, S.M.: A same/different fault dictionary: An extended pass/fail fault dictionary with improved diagnostic resolution. In: DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1474–1479 (2008)
24. Richman, J., Bowden, K.R.: The modern fault dictionary. In: Proceedings of the International Test Conference, pp. 696–702 (1985)
25. Tai, S., Bhattacharya, D.: Pipelined fault simulation on parallel machines using the circuitflow graph. In: Computer Design: VLSI in Computers and Processors, pp. 564–567 (1993)
26. Tulloss, R.E.: Fault dictionary compression: Recognizing when a fault may be unambiguously represented by a single failure detection. In: Proceedings of the Test Conference, pp. 368–370 (1980)

Chapter 10
Accelerating Circuit Simulation Using Graphics Processors

10.1 Chapter Overview

SPICE [14] based circuit simulation is a traditional workhorse in the VLSI design process. Given the pivotal role of SPICE in the IC design flow, there has been significant interest in accelerating SPICE. Since a large fraction (on average 75%) of the SPICE runtime is spent in evaluating transistor model equations, a significant speedup can be availed if these evaluations are accelerated. This chapter reports on our efforts to accelerate transistor model evaluations using a graphics processing unit (GPU). We have integrated this accelerator with OmegaSIM, a commercial fast SPICE [6] tool. Our experiments demonstrate that significant speedups (2.36× on average) can be obtained. The asymptotic speedup that can be obtained is about 4×. We demonstrate that with circuits consisting of as few as about 1,000 transistors, speedups of ∼3× can be obtained. By utilizing NVIDIA Tesla GPU systems [5], this speedup could be enhanced further, especially for larger designs.

The remainder of this chapter is organized as follows. Section 10.2 introduces circuit simulation along with the motivation to accelerate it. Some previous work in circuit simulation has been described in Section 10.3. Section 10.4 details our approach for implementing device model evaluation on a GPU. In Section 10.5 we present results from experiments which were conducted after implementing our approach and integrating it in OmegaSIM. We summarize the chapter in Section 10.6.

10.2 Introduction

SPICE [14] is the de facto industry standard for circuit-level simulation of VLSI designs. SPICE simulation is typically infeasible for designs larger than 20,000 devices. With the rapidly decreasing minimum feature sizes of devices, the number of devices on a single chip has significantly increased. As a result, it becomes critically important to run SPICE on larger portions of the design to validate their electrical and timing behavior before tape-out. Further, process variations increasingly impact the electrical behavior of a design. This is often tackled by performing Monte Carlo SPICE simulations, requiring significant computing and time resources.

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_10,
© Springer Science+Business Media, LLC 2010

As a result, there is a significant motivation to speed up SPICE simulations without losing accuracy. In this chapter, we present an approach to accelerate the computationally intensive component of SPICE, by exploiting the parallelism available in graphics processing units (GPUs). In particular, our approach parallelizes and accelerates the transistor model evaluation in SPICE, for BSIM3 [1] models. Our benchmarking shows that BSIM3 model evaluations comprise about 75% of the SPICE runtime. By accelerating this portion of SPICE, therefore, a speedup of up to 4× can be obtained in theory. Our results show that in practice, our approach can obtain a speedup of about 2.36× on average, with a maximum speedup of 3.07×. The significance of this is further underscored by the fact that our approach is implemented and integrated in OmegaSIM [6], a commercial SPICE accelerator tool, which presents significant speed gains over traditional SPICE implementations, even without GPU-based acceleration.

The SPICE algorithm and its variants simulate the non-linear time-varying behavior of a design, by employing the following key procedures:

- formulation of circuit equations using modified nodal analysis [16] (MNA) or sparse tableau analysis [13] (STA);
- evaluating the time-varying behavior of the design using numerical integration techniques, applied to the non-linear circuit model;
- solving the non-linear circuit model using Newton–Raphson (NR) based iterations; and
- solving a linear system of equations in the inner loop of the engine.

The main time-consuming computation in SPICE is the evaluation of device model equations in different iterations of the above flow. Our profiling experiments, using BSIM3 models, show that on average 75% of the SPICE runtime is spent in performing these evaluations. This is because these evaluations are performed for each device and possibly repeated for each time step, until the convergence of the NR-based non-linear equation solver. The total number of such evaluations can easily run into the billions, even for small- to medium-sized designs. Therefore, the speed of the device model evaluation code is a significant determinant of the speed of the overall SPICE simulator [16]. For more accurate device models like BSIM4 [2], which account for additional electrical behaviors of deep sub-micron (DSM) devices, the fraction of the total runtime which model evaluations require is even higher. Thus the asymptotic speedup that can be obtained by accelerating these evaluations is more than 4×.

This chapter focuses on the acceleration of SPICE by performing the transistor model evaluations on the GPU. An industrial design could require several thousand device model evaluations for a given time step. These evaluations are independent. In other words the device model computation requires that the same set of model equations be evaluated, possibly several thousand times, for different devices with no data dependencies. This property of the device model evaluations matches well with the single instruction multiple data (SIMD) computational paradigm that GPUs implement. Our current implementation handles BSIM3 models. However, using

the approach described in the chapter, we can easily handle BSIM4 models or a combination of different models.

Our device model evaluation engine is implemented in the Compute Unified Device Architecture (CUDA) framework, which is an open-source programming and interfacing tool provided by NVIDIA for programming their GPU devices. The GPU device used for our implementation and benchmarking is the NVIDIA GeForce 8800 GTS.

Performing the evaluation of device model equations for several thousand devices is a natural match for capabilities of the GPU. This is because such an application can exploit the extreme memory bandwidths of the GPU, as well as the presence of several computation elements on the GPU. To the best of the authors' knowledge, this work is the first to accelerate circuit simulation on a GPU platform.

An extended abstract of this work can be found in [12]. The key contributions of this work are as follows:

- We *exploit the match between parallel device model evaluation and the capabilities of a GPU, a SIMD-based device*. This enables us to harness the computational power of GPUs to accelerate device model evaluations.
- Our implementation caters to the key features required to obtain maximal speedup on a GPU:
  - The different threads, which perform device model evaluations, are implemented so that there are no data or control dependencies between threads.
  - All device model evaluation threads compute identical instructions, but on different data, which exploits the SIMD architecture of the GPU.
  - The values of the device parameters required for evaluating the model equations are obtained using a texture memory lookup, thus exploiting the extremely large memory bandwidth of GPUs.
- Our device model evaluation is implemented in a manner which is aware of the specific constraints of the GPU platform such as the use of (cached) texture memory for table lookup, memory coalescing for global memory accesses, and the balancing of hardware resources used by different threads. This helps maximize the speedup obtained.
- Our approach is integrated into a commercial circuit simulation tool OmegaSIM [6]. A CPU-only implementation of OmegaSIM is on average 10–1,000× faster than SPICE (and about 10× faster than other fast SPICE implementations). With the device model evaluation performed using our GPU-based implementation, OmegaSIM is sped up by an additional factor of 2.36×, on average.

10.3 Previous Work

Several fast SPICE implementations depend upon hierarchical isomorphism to increase performance [7, 4, 3]. In other words they extract hierarchical similarities in the design and avoid redundant simulations. This approach works well for regular

designs such as memories, which exhibit a high degree of repetitive and hierarchical structure. However, it is less successful for random logic or other designs without repetitive structures. This approach is not efficient for simulating a post place-and-routed design, since back-annotated capacitances vary significantly so that repetitive blocks of hierarchy can no longer be considered to be identical in terms of their electrical behavior. Our approach parallelizes device model evaluations at each time step and hence exhibits a healthy speedup regardless of the regularity (or lack thereof) in a circuit. As such, our approach is orthogonal to the hierarchical isomorphism-based techniques.

A transistor-level engine targeted for interconnect analysis is proposed in [11]. It makes use of the successive chord (SC) integration method (as opposed to NR iterations) and a table lookup model to determine I_{ds} currents. The approach reuses LU factorization results across multiple time steps and input stimuli. As noted by the authors, the SC method does not provide all desired convergence properties of the NR method for general analog simulation analysis. In contrast, our approach speeds up device model evaluation for arbitrary circuits in a classical SPICE framework, due to its robustness and industry-wide popularity. Our early experiments demonstrate that model evaluation comprises the majority ($\sim$75%) of the total circuit simulation runtime. Our approach is orthogonal to the non-linear system solution approach and can thus be used in tandem with the approach of [11] if desired.

The approach of [8] proposed speeding up device model evaluation by using the PACE [9] distributed memory multiprocessor system, with a four-processor cluster. They targeted transient analysis in ADVICE, an AT&T circuit simulation program similar to SPICE, which is available commercially. Our approach, in contrast, exploits the parallelism available in an off-the-shelf GPU for speeding up device model evaluations. Further, their experimental results discuss the speedup obtained for device model evaluation (*alone*) to be about 3.6×. Our results speed up device model evaluation by 30–40× on average. The speedup obtained using our approach for the *entire* SPICE simulation is 2.36× on average. Further, their target multiprocessor system requires the user to perform load balancing up-front. The CUDA architecture and its instruction scheduler (which handles the GPU memory accesses) together abstract the problem of load balancing away from the user. Also, the thread scheduler on the GPU periodically switches between processors to efficiently and dynamically balance their computational resources, without user intervention.

The authors of [17] proposed speeding up circuit simulation using a shared memory multiprocessor system, namely the Alliant FX/8 with a six-processor cluster. They too target transient analysis in ADVICE, but concentrate on two routines – (i) an implicit numerical integration scheme to solve the time-varying non-linear system and (ii) a modified approach for solving the set of non-linear equations. In contrast, our approach uses a commercial off-the-shelf GPU to accelerate only the device model evaluations, by exploiting the SIMD computing paradigm of the GPU. During numerical integration, the authors perform device model evaluation by device type. In other words, all resistors are evaluated at once, then all capacitors are evaluated followed by MOSFETs, etc. In order to avoid potential conflicts due to parallel writes, the authors make use of locks for consistency. Our implementation

faces no such issues, since all writes are automatically synchronized by the scheduler and are thus conflict-free. Therefore, we obtain significantly higher speedups. The experimental results of [17] indicate a speedup for *device model evaluation* of about 1–6×. Our results demonstrate speedups for device model evaluation of about 30–40×. The authors of [17] do not report runtimes or speedup obtained for the entire circuit simulation. We improve the runtime for the complete circuit simulation by 2.36× on average.

The commercial tool we used for integrating our implementation of GPU-based device model evaluation is OmegaSIM [6]. OmegaSIM's core is a multi-engine, current-based architecture with multi-threading capabilities. Other details about the OmegaSIM architecture are not pertinent to this chapter, since we implement only the device model evaluations on the GPU.

10.4 Our Approach

The SPICE [15, 14] algorithm simulates the non-linear time-varying behavior of a circuit using the following steps:

- First, the circuit equations are formulated using modified nodal analysis (MNA). This is typically done by *stamping* the MNA matrix based on the types of devices included in the SPICE netlist, as well as their connectivity.
- The time-varying behavior of the design is solved using numerical integration techniques applied to the non-linear circuit model. Typically, the trapezoidal method of numerical integration is used, although the Gear method may be optionally used. Both these methods are implicit methods and are highly stable.
- The non-linear circuit model is solved using Newton–Raphson (NR) based iterations. In each iteration, a linear system of equations needs to be solved. During the linearization step, device model equations need to be evaluated, to populate the coefficient values in the linear system of equations.
- Solving a linear system of equations forms the inner loop of the SPICE engine.

We profiled the SPICE code to find the fraction of time that is spent performing device model evaluations, on several circuits. These profiling experiments, which were performed using OmegaSIM, showed that on average 75% of the total simulation runtime is spent in performing device model evaluations for industrial designs. As an example, for the design *Industry_1*, which performs the functionality of a Linear Feedback Shift Register (LFSR), 74.9% of the time was spent in BSIM3 device model evaluations. The *Industry_1* design had 324 devices and required 1.86×10^7 BSIM3 device model evaluations over the entire simulation.

We note that the device model evaluations can be performed in parallel, since they need to be evaluated for every device. Further, these evaluations are possibly repeated (with different input data) for each time step until the convergence of the NR-based non-linear equation solver. Therefore, billions of these evaluations could be required for a complete simulation, even for small to medium designs. Also,

these computations are independent of each other, exposing significant parallelism for medium- to large-sized designs. The speed of execution of the device model evaluation code, therefore, significantly determines the speed of the overall SPICE simulator. Since the GPU platform allows significant parallelism, it forms an ideal candidate platform for speeding up transistor model evaluations. Since device model evaluations consume about 75% of the runtime of a CPU-based SPICE engine, we can obtain an asymptotic maximum speedup of 4× if these computations are parallelized. This is in accordance with Amdahl's law [10], which states that the overall algorithm performance is limited by the portion that is not parallelizable. In the sequel we discuss the implementation of the GPU-based device model evaluation portion of the SPICE flow.

Our implementation is integrated into an industrial accelerated SPICE tool called OmegaSIM. Note that OmegaSIM, running in a CPU-only mode, obtains significant speedup over competing SPICE offerings. Our implementation, after integration into OmegaSIM, results in a CPU+GPU implementation which is 2.36× faster on average, compared to the CPU-only version of OmegaSIM.

10.4.1 Parallelizing BSIM3 Model Computations on a GPU

Our implementation supports BSIM3 models. In this section, we make several observations about the careful engineering required in order to parallelize BSIM3 device model computations on a GPU. These ideas are implemented in our approach and together help us achieve the significant speedup in BSIM3 model computations. Note that BSIM4 device model computations can be parallelized in a similar manner.

10.4.1.1 Inlining if–then–else Code

The BSIM3 model evaluation code consists of several *if–then–else* statements, with a maximum nesting depth of 4. This code does not contain any *while* or *for* loops. The input to the BSIM3 device model evaluation routine is a number of device parameters, some of which are unchanged during the model evaluation (these parameters are referred to as *runtime parameters*), while others are computed during the model evaluation. The key task is to perform these computations on a GPU, which has a SIMD computation model. For instance, a code fragment such as

```
   Codefragment1()
if(cond){ CODE-A; }
else{ CODE-B; }
```

would be converted into the following code fragment for execution on the GPU:

```
   Codefragment2()
CODE-A;
CODE-B;
if(cond){ return result of CODE-A; }
else{ return result of CODE-B; }
```

As mentioned, the programming paradigm of a GPU is the single instruction multiple data (SIMD) model, wherein all threads must compute identical instructions, but on different data. The different threads being computed in parallel should have no data or control dependency among them, to obtain maximal speedup. GPUs also have an extremely large memory bandwidth, which allows multiple memory accesses to be performed in parallel. The SIMD paradigm is thus an appropriate match for performing several device model evaluations in parallel. Our code (restructured as shown in *Codefragment2()*) can be executed in a SIMD fashion on a GPU, with all kernels executing the same instruction in lock-step, but on different data. Of course, this code fragment requires the GPU to perform more instructions than is the case with the original code fragment. However, the large degree of parallelism on the GPU overcomes this disadvantage and yields impressive speedups, as we will see in the sequel. The above conversion is handled by the CUDA compiler.

10.4.1.2 Partitioning the BSIM3 Code into Kernels

The key task in implementing the BSIM3 device model evaluations on the GPU is the partitioning of the BSIM3 code into smaller fragments, with each fragment being implemented as a GPU kernel.

In the limit, we could implement the entire BSIM3 code in a single kernel, which includes all the device model evaluations required for a BSIM3 model card. However, this would not allow us to execute a sufficiently large number of kernels in parallel. This is because of the limitation on the hardware resources available for every multiprocessor on the GPU. In particular, the limitation applies to registers and shared memory. As mentioned earlier, the maximum number of registers for a multiprocessor is 8,192. Also, the maximum amount of shared memory for a multiprocessor is 16 KB. If any of these resources are exceeded, additional kernels cannot be run. Therefore, if we had a kernel with 4,000 registers, then no more than 2 kernels can be issued in parallel (even if the amount of shared memory used by these 2 kernels is much less than 16 KB). In order to achieve maximal speedup, the GPU code needs to be implemented in a manner that hides memory access latencies, by issuing hundreds of threads at once. In case a single thread (which implements all the device model evaluations) is launched, it will not leave sufficient hardware resources to instantiate a sufficient number of additional threads to execute the same kernel (on different data). As a result, the latency of accessing off-chip memory will not be hidden in such a scenario. To avert this, the device model evaluation code needs to be partitioned into smaller kernels. These kernels are of an appropriate size such that a large number of them can be issued without depleting the registers or shared memory of the multiprocessor. If, on the other hand, the kernels are too small, then large amounts of data transfer will be required from one kernel to another (this transfer is done via global memory). The data that is written by kernel k, and needs to be read by kernel $k + j$, will be stored in global memory. If the kernels are extremely small, a large amount of data will have to be written and read to/from global memory, hampering the performance. Hence, in the other extreme case of very small kernels, we may run into a performance bottleneck as well.

Therefore, keeping in mind the limited hardware resources (in terms of registers and shared memory), and the global memory latency and bandwidth constraints, the device model evaluations are partitioned into appropriately sized kernels which maximize parallelism and minimize the global memory access latency. Satisfying both these constraints for a kernel is important in order to maximally exploit the speedup obtained on the GPU.

Our approach for partitioning the BSIM3 code into kernels first finds the control and dataflow graph (CDFG) of the BSIM3 code. Then we find the disconnected components of this graph, which form a set D. For each component $d \in D$, we partition the code of d into smaller kernels as appropriate. The partitioning is performed such that the number of variables that are written by kernel k and read by kernel $k + j$ is minimized. This minimizes the number of global memory accesses. Also, the number of registers R used by each kernel is minimized, since the total number of threads that can be issued in parallel on a single multiprocessor is $8{,}192/R$, rounded down to the nearest multiple of 32, as required by the 8800 architecture. The number of threads issued in parallel cannot exceed 768 for a single multiprocessor.

10.4.1.3 Efficient Use of GPU Memory Model

In order to obtain maximum speedup of the BSIM3 model evaluation code, the different forms of GPU memory need to be carefully utilized. In this section, we discuss the approach taken in this regard:

- **Global Memory**
  At a coarse analysis level, the device model evaluations in a circuit simulator are divided into

  – creating a DC model for the device, given the operating voltages at the device terminals, and
  – calculating the different output values that are part of the BSIM3 device evaluation code. These are the values that are returned by the BSIM3 device evaluation code, to the calling routine.

  In order to minimize the data transfers from GPU (device) to CPU (host), the results of the set of kernels that compute the DC model parameters are stored in global memory and are not returned back to the host. Next, when the kernels which calculate the values that need to be returned by the BSIM3 model evaluation routine are executed, they can read (or write) the global memory to fetch the DC model parameters. GPUs have an extremely large memory bandwidth as discussed earlier, which allows multiple memory accesses to the global memory to be performed in parallel and their latencies to be hidden.
- **Texture Memory**
  In our implementation, the values of the parameters (referred to as *runtime parameters*) required for performing device model evaluations are stored in the texture memory and are accessed by performing a texture memory lookup. Using

the texture memory (as opposed to global, shared, or constant memory) has the following advantages:

- Texture memory of a GPU device is cached as opposed to shared or global memory. Hence we can exploit the benefits obtained from the cached texture memory lookups.
- Texture memory accesses do not have coalescing constraints as required in case of global memory accesses, making the runtime parameters lookup efficient.
- In case of multiple lookups performed in parallel, shared memory accesses might lead to bank conflicts and thus impede the potential speedup.
- Constant memory accesses in the GPU, as discussed in Chapter 3, are optimal when all lookups occur at the same memory location. This is typically not the case in parallel device model evaluation.
- The CUDA programming environment has built-in texture fetching routines which are extremely efficient.

Note that the allocation and loading of the texture memory requires non-zero time, but this cost is easily amortized since several thousand lookups are typically performed from the same runtime parameter data.

- **Constant Memory**
  Our implementation makes efficient use of the constant memory for storing physical constants such as π, ε_o required for device model evaluations. Constant memory is cached, and thus, performing several million device model evaluations in the entire circuit simulation flow allows us to exploit the advantage of a cached constant memory. Since the processors in any multiprocessor of the GPU operate in a SIMD fashion, all lookups for the constant parameters occur at the same memory location at the same time. This results in the most optimal usage of constant memory.

10.4.1.4 Thread Scheduler and Code Statistics

Once the threads are issued to the GPU, an inbuilt hardware scheduler performs the scheduling of these threads.

The blocks that are processed by one multiprocessor in one batch are referred to as active. Each active block is split into SIMD groups of threads called warps. Each of these warps contains the same number of threads (warp size) and is executed by the multiprocessor in a SIMD fashion. Active warps (i.e., all the warps from all the active blocks) are *time-sliced* – a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources.

The statistics for our implementation of the BSIM3 device model evaluation code are reported next. The warp size for an NVIDIA 8800 device is 32. Further, the pool of registers available for the threads in a single multiprocessor is equal to 8,192. In our implementation, the *dimblock* size is 32 threads. The average number of registers used by a single kernel in our implementation is around 12. A register

count limit of 8,192 allows 640 threads of the possible 768 threads in a single multiprocessor to be issued, thus having an occupancy of about 83.33% on average. The multiprocessor occupancy is calculated using the *occupancy calculator worksheet* provided with CUDA. The number of registers used per kernel and the shared memory per block are obtained using the CUDA compiler (*nvcc*) with the *'-cubin'* option.

10.5 Experiments

Our device model evaluation engine is implemented and integrated in a commercial SPICE accelerator tool OmegaSIM [6]. In this section, we present details of our experimental setup and results.

In all our experiments, the CPU used was an Intel Core 2 Quad Core (4 processor) machine, running at 2.40 GHz with 4 GB RAM. The GPU card used for our experiments is the NVIDIA GeForce 8800 GTS with 512 MB RAM, operating at 675 MHz.

We first profiled the circuit simulation code. Over several examples, we found that about 75% of the runtime is consumed by BSIM3 device model evaluations. For the design *Industrial_1*, the code profiling is as follows:

- BSIM3 device model evaluations = 74.9%.
- Non-accelerated portion of OmegaSIM code = 24.1%.

Thus, by accelerating the BSIM3 device evaluation code, we can asymptotically obtain around 4× speedup for circuit simulation.

Table 10.1 compares our approach of implementing the device model evaluation on the GPU with the device model evaluation on the CPU in terms of runtime. Column 1 reports the number of evaluations performed. Columns 2 and 3 report the GPU runtimes (wall clock), in ms, for evaluating the device model equations and the data transfers (CPU → GPU as well as GPU → CPU), respectively. In particular, the data transfers include transferring the *runtime parameters* and the operating voltages at the device terminal (for all evaluations) from CPU to GPU. The data transfers from the GPU to CPU include the outputs of the BSIM3 device model evaluation code. Column 4 reports the total (processing+transfer) GPU runtimes (in ms). The CPU runtimes (in ms) are reported in column 5 and the speedup obtained is reported in column 6.

Table 10.1 Speedup for BSIM3 evaluation

	GPU runtimes (ms)				
# Evaluations	Processing	Transfer	Total	CPU runtime (ms)	Speedup
1M	81.17	196.48	277.65	8,975.63	32.33 ×
2M	184.91	258.79	443.7	18,086.29	40.76 ×

Table 10.2 Speedup for circuit simulation

			OmegaSIM (s)	AuSIM (s)	
Ckt name	# Trans.	Total # eval.	CPU-alone	GPU+CPU	SpeedUp
Industrial_1	324	1.86×10^7	49.96	34.06	1.47 ×
Industrial_2	1,098	2.62×10^9	118.69	38.65	3.07 ×
Industrial_3	1,098	4.30×10^8	725.35	281.5	2.58 ×
Buf_1	500	1.62×10^7	27.45	20.26	1.35 ×
Buf_2	1,000	5.22×10^7	111.5	48.19	2.31 ×
Buf_3	2,000	2.13×10^8	486.6	164.96	2.95 ×
ClockTree_1	1,922	1.86×10^8	345.69	132.59	2.61 ×
ClockTree_2	7,682	1.92×10^8	458.98	182.88	2.51 ×
Avg					2.36 ×

Table 10.2 compares the runtime of AuSIM (which is OmegaSIM with our approach integrated. AuSIM runs partly on GPU and partly on CPU against the original OmegaSIM (running on the CPU alone). Columns 1 and 2 report the circuit name and the number of transistors in the circuit, respectively. The number of evaluations required for full circuit simulation is reported in column 3. Columns 4 and 5 report the CPU-alone and GPU+GPU runtimes (in seconds), respectively. The speedups are reported in column 6. The circuits Industrial_1, Industrial_2, and Industrial_3 perform the functionality of an LFSR. Circuits Buf_1, Buf_2, and Buf_3 are buffer insertion instances for buses of three different sizes. Circuits ClockTree_1 and ClockTree_2 are symmetrical H-tree clock distribution networks. These results show that an average speedup of 2.36× can be achieved over a variety of circuits. Also, note that with an increase in the number of transistors in the circuit, the speedup obtained is higher. This is because the GPU memory latencies can be better hidden when more device evaluations are issued in parallel.

The NVIDIA 8800 GPU device supports IEEE 754 single precision floating point operations. However, the BSIM3 model code uses IEEE 754 double precision floating point computations. We first converted all the double precision computations in the BSIM3 code into single precision before modifying it for use on the GPU. We determined the error that was incurred in this process. We found that the accuracy obtained by our GPU-based implementation of device model evaluation (using single precision floating point) is extremely close to that of a CPU-based double precision floating point implementation. In particular, we computed the error over 10^6 device model evaluations and found that the maximum absolute error was 9.0×10^{-22} Amperes, and the average error was 2.88×10^{-26} Amperes. The relative average error was 4.8×10^{-5}. NVIDIA has announced the availability of GPU devices which support double precision floating point operations. Such devices will further improve the accuracy of our approach.

Figures 10.1 and 10.2 show the voltage plots obtained for *Industrial*_2 and *Industrial*_3 circuits, obtained by running AuSIM and comparing it with SPICE. Notice that the plots completely overlap.

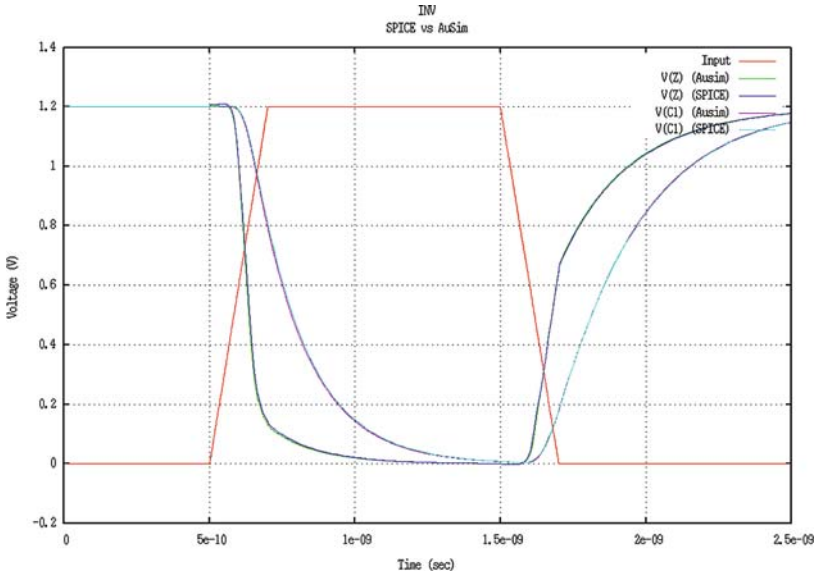


Fig. 10.1 Industrial_2 waveforms

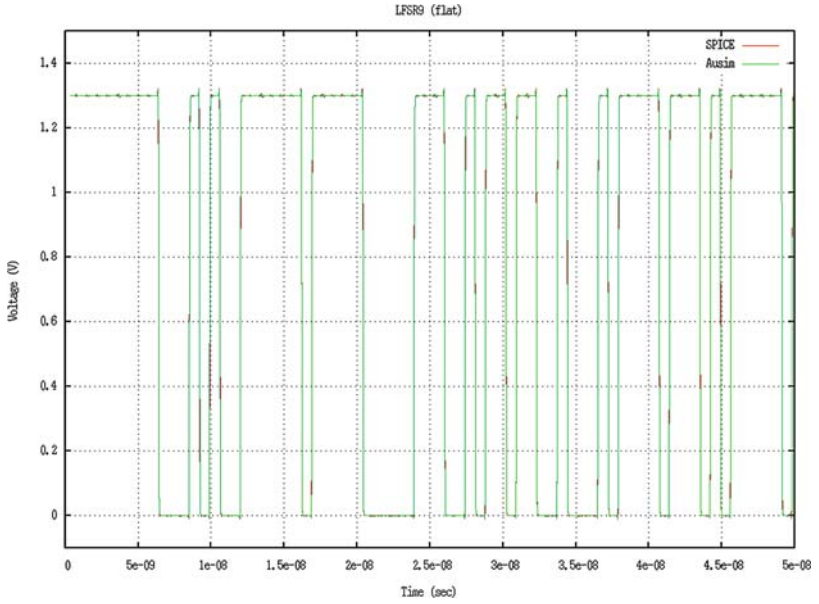


Fig. 10.2 Industrial_3 waveforms

10.6 Chapter Summary

Given the key role of SPICE in the design process, there has been significant interest in accelerating SPICE. A large fraction (on average 75%) of the SPICE runtime is spent in evaluating transistor model equations. The chapter reports our efforts to accelerate transistor model evaluations using a GPU. We have integrated this accelerator with a commercial fast SPICE tool and have shown significant speedups (2.36× on average). The asymptotic speedup that can be obtained is about 4×. With the recently announced quad GPU systems, this speedup could be enhanced further, especially for larger designs.

References

1. BSIM3 Homepage. http://www-device.eecs.berkeley.edu/~bsim3
2. BSIM4 Homepage. http://www-device.eecs.berkeley.edu/~bsim4
3. Capsim Hierarchical Spice Simulation. http://www.xcad.com/xcad/spice-simulation.html
4. FineSIM SPICE. http://www.magmada.com/c/SVX0QdBvGgqX/Pages/FineSimSPICE.html
5. NVIDIA Tesla GPU Computing Processor. http://www.nvidia.com/object/IO_43499.html
6. OmegaSim Mixed-Signal Fast-SPICE Simulator. http://www.nascentric.com/product.html
7. Virtuoso UltraSim Full-chip Simulator. http://www.cadence.com/products/custom_ic/ultrasim/index.aspx
8. Agrawal, P., Goil, S., Liu, S., Trotter, J.: Parallel model evaluation for circuit simulation on the PACE multiprocessor. In: Proceedings of the Seventh International Conference on VLSI Design, pp. 45–48 (1994)
9. Agrawal, P., Goil, S., Liu, S., Trotter, J.A.: PACE: A multiprocessor system for VLSI circuit simulation. In: Proceedings of SIAM Conference on Parallel Processing, pp. 573–581 (1993)
10. Amdahl, G.: Validity of the single processor approach to achieving large-scale computing capabilities. Proceedings of AFIPS **30**, 483–485 (1967)
11. Dartu, F., Pileggi, L.T.: TETA: transistor-level engine for timing analysis. In: DAC '98: Proceedings of the 35th Annual Conference on Design Automation, pp. 595–598 (1998)
12. Gulati, K., Croix, J., Khatri, S.P., Shastry, R.: Fast circuit simulation on graphics processing units. In: Proceedings, IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC), pp. 403–408 (2009)
13. Hachtel, G., Brayton, R., Gustavson, F.: The sparse tableau approach to network analysis and designation. Circuits Theory, IEEE Transactions on **18**(1), 101–113 (1971)
14. Nagel, L.: SPICE: A computer program to simulate computer circuits. In: University of California, Berkeley UCB/ERL Memo M520 (1995)
15. Nagel, L., Rohrer, R.: Computer analysis of nonlinear circuits, excluding radiation. IEEE Journal of Solid States Circuits **SC-6**, 162–182 (1971)
16. Pillage, L.T., Rohrer, R.A., Visweswariah, C.: Electronic Circuit & System Simulation Methods. McGraw-Hill, New York (1994). ISBN-13: 978-0070501690 (ISBN-10: 0070501696)
17. Sadayappan, P., Visvanathan, V.: Circuit simulation on shared-memory multiprocessors. IEEE Transactions on Computers **37**(12), 1634–1642 (1988)

Part IV
Automated Generation of GPU Code

Outline of Part IV

In Part I of this monograph candidate hardware platforms were discussed. In Part II, we presented three approaches (custom IC based, FPGA based, and GPU-based) for accelerating Boolean satisfiability, a control-dominated EDA application. In Part III, we presented the acceleration of several EDA applications with varied degrees of inherent parallelism in them. In Part IV of this monograph, we present an automated approach to accelerate uniprocessor code using a GPU. The key idea here is to partition the software application into kernels in an automated fashion, such that multiple instances of these kernels, when executed in parallel on the GPU, can maximally benefit from the GPU's hardware resources.

Due to the high degree of available hardware parallelism on the GPU, these platforms have received significant interest for accelerating scientific software. The task of implementing a software application on a GPU currently requires significant manual effort (porting, iteration, and experimentation). In Chapter 11, we explore an automated approach to partition a uniprocessor software application into kernels (which are executed in parallel on the GPU). The input to our algorithm is a uniprocessor subroutine which is executed multiple times, on different data, and needs to be accelerated on the GPU. Our approach aims at automatically partitioning this routine into GPU kernels. This is done by first extracting a graph which models the data and control dependencies of the subroutine in question. This graph is then partitioned. Various partitions are explored, and each is assigned a cost which accounts for GPU hardware and software constraints, as well as the number of instances of the subroutine that are issued in parallel. From the least cost partition, our approach automatically generates the resulting GPU code. Experimental results demonstrate that our approach correctly and efficiently produces fast GPU code, with high quality. We show that with our partitioning approach, we can speed up certain routines by 15% on average when compared to a monolithic (unpartitioned) implementation. Our entire technique (from reading a C subroutine to generating the partitioned GPU code) is completely automated and has been verified for correctness.

Chapter 11
Automated Approach for Graphics Processor Based Software Acceleration

11.1 Chapter Overview

Significant manual design effort is required to implement a software routine on a GPU. This chapter presents an automated approach to partition a software application into kernels (which are executed in parallel) that can be run on the GPU. The software application should satisfy the constraint that it is executed multiple times on different data, and there exist no control dependencies between invocations. The input to our algorithm is a C subroutine which needs to be accelerated on the GPU. Our approach automatically partitions this routine into GPU kernels. This is done as follows. We first extract a graph which models the data and control dependencies of the target subroutine. This graph is then partitioned using a K-way partition, using several values of K. For every partition a cost is computed which accounts for GPU's hardware and software constraints. The cost also accounts for the number of instances of the subroutine that are issued in parallel. We then select the least cost partitioning solution and automatically generate the resulting GPU code corresponding to this partitioning solution. Experimental results demonstrate that our approach correctly and efficiently produces high-quality, fast GPU code. We demonstrate that with our partitioning approach, we can speed up certain routines by 15% on average, when compared to a monolithic (unpartitioned) implementation. Our approach is completely automated and has been verified for correctness.

The remainder of this chapter is organized as follows. The motivation for this work is described in Section 11.2. Section 11.3 details our approach for kernel generation for a GPU. In Section 11.4 we present results from experiments and summarize in Section 11.5.

11.2 Introduction

There are typically two broad approaches that have been employed to accelerate scientific computations on the GPU platform. The first approach is the most common and involves taking a scientific application and *rearchitecting* its code to exploit the GPU's capabilities. This redesigned code is now run on the GPU. Significant

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_11,
© Springer Science+Business Media, LLC 2010

speedup has been demonstrated in this manner, for several algorithms. Examples of this approach include the GPU implementations of sorting [9], the map-reduce algorithm [4], and database operations [3]. A good reference in this area is [8].

The second approach involves identifying a particular subroutine S in a CPU-based algorithm (which is repeated multiple times in each iteration of the computation and is found to take up a majority of the runtime of the algorithm) and accelerating it on the GPU. We refer to this approach as the *porting* approach, since only a portion of the original CPU-based code is ported on the GPU without any rearchitecting of the code. This approach requires less coding effort than the rearchitecting approach. The overall speedup obtained through this approach is, however, subject to Amdahl's law, which states that if a parallelizable subroutine which requires a fractional runtime of P is sped up by a factor Q, then the final speedup of the overall algorithm is

$$\frac{1}{(1-P)+\frac{P}{Q}} \tag{11.1}$$

The rearchitecting approach typically requires a significant investment of time and effort. The porting approach is applicable to many problems in which a small number of subroutines are run repeatedly on independent data values and take up a large fraction of the total runtime. Therefore, an approach to automatically generate GPU code for such problems would be very useful in practice.

In this chapter, we focus on automatically generating GPU code for the porting class of problems. Porting implementations require careful partitioning of the subroutine into kernels which are run in parallel on the GPU. Several factors must be considered in order to come up with an optimal solution:

- To maximize the speedup obtained by executing the subroutine on the GPU, numerous and sometimes conflicting constraints imposed by the GPU platform must be accounted for. In fact, if a given subroutine is run without considering certain key constraints, the subroutine may fail to execute on the GPU altogether.
- The number of kernels and the total communication and computation costs for these kernels must be accounted for as well.

Our approach partitions the program into kernels, multiple instances of which are executed (on different data) in parallel on the GPU. Our approach also schedules the partitions in such a manner that correctness is retained. The fact that we operate on a restricted class of problems[1] and a specific parallel processing platform (the GPU) makes the task of automatically generating code more practical. In contrast the task of general parallelizing compilers is significantly harder, There has been significant research in the area of parallelizing compilers. Examples include the Parafrase Fortran reconstructing compiler [6]. Parafrase is an optimizing compiler preprocessor

[1] Our approach is employed for subroutines that are executed multiple times, on independent data.

that takes as input scientific Fortran code, constructs a program dependency graph, and performs a series of optimization steps that creates a revised version of the original program. The automatic parallelization targeted in [6] is limited to the loops and array references in numeric applications. The resultant code is optimized for multiple instruction multiple data (MIMD) and very long instruction word (VLIW) architectures. The Bulldog Fortran reassembling compiler [2] is aimed at automatic parallelization at the instruction level. It is designed to detect parallelism that is not amenable to vectorization by exploiting parallelism within the basic block.

The key contrasting features of our approach to existing parallelizing compilers are as follows. First, our target platform is a GPU. Thus the constraints we need to satisfy while partitioning code into kernels arise due to the hardware and architectural constraints associated with the GPU platform. The specific constraints are detailed in the sequel. Also, the memory access patterns required for optimized execution of code on a GPU are very specific and quite different from a general vector or multi-core computer. Our approach attempts to incorporate these requirements while generating GPU kernels automatically.

11.3 Our Approach

Our kernel generation engine automatically partitions a given subroutine S into K kernels in a manner that maximizes the speedup obtained by multiple invocations of these kernels on the GPU. Before our algorithm is invoked, the key decision to be made is the determination of which subroutine(s) to parallelize. This is determined by profiling the program and finding the set of subroutines Σ that

- are invoked repeatedly and independently (with different input data values) and
- collectively take up a large fraction of the runtime of the entire program. We refer to this fraction as P.

Now each subroutine $S \in \Sigma$ is passed to our kernel generation engine, which automatically generates the GPU kernels for S.

Without loss of generality, in the remainder of this section, our approach is described in the context of kernel generation for a single subroutine S.

11.3.1 Problem Definition

The goal of our kernel generation engine for GPUs is stated as follows. Given a subroutine S and a number N which represents the number of independent calls of S that are issued by the calling program (on different data), find the best partitioning of S into kernels, for maximum speedup when the resulting code is run on a GPU.

In particular, in our implementation, we assume that S is implemented in the C programming language, and the particular SIMD machine for which the kernels are generated is an NVIDIA Quadro 5800 GPU. Note that our kernel generation

engine is general and can generate kernels for other GPUs as well. If an alternate GPU is used, this simply means that the cost parameters to our engine need to be modified. Also, our kernel generation engine handles in-line code, nested if–then–else constructs of arbitrary depth, pointers, structures, and non-recursive function calls (by value).

11.3.2 GPU Constraints on the Kernel Generation Engine

In order to maximize performance, GPU kernels need to be generated in a manner that satisfies constraints imposed by the GPU-based SIMD platform. In this section, we summarize these constraints. In the next section, we describe how these constraints are incorporated in our automatic kernel generation engine:

- As mentioned earlier, the NVIDIA Quadro 5800 GPU consists of 30 multiprocessors, each of which has 8 processors. As a result, there are 240 hardware processors in all, on the GPU IC. For maximum hardware utilization, it is important that we issue significantly more than 240 threads at once. By issuing a large number of threads in parallel, the data read/write latencies of any thread are hidden, resulting in a maximal utilization of the processors of the GPU, and hence ensuring maximal speedup.
- There are 16,384 32-bit registers per multiprocessor. Therefore if a subroutine S is partitioned into K kernels, with the ith kernel utilizing r_i registers, then we should have $\max_i(r_i) \cdot$ (# of threads per MP) $\leq$ 16,384. This argues that across all our kernels, if $\max_i(r_i)$ is too small, then registers will not be completely utilized (since the number of threads per multiprocessor is at most 1,024), and kernels will be smaller than they need to be (thereby making K larger). This will increase the communication cost between kernels.
  On the other hand, if $\max_i(r_i)$ is very high (say 4,000 registers for example), then no more than 4 threads can be issued in parallel. As a result, the latency of accessing off-chip memory will not be hidden in such a scenario. In the CUDA programming model, if r_i for the ith kernel is too large, then the kernel fails to launch. Therefore, satisfying this constraint is important to ensure the execution of any kernel. We try to ensure that r_i is roughly constant across all kernels.
- The number of threads per multiprocessor must be
  - a multiple of 32 (since 32 threads are issued per warp, the minimum unit of issue),
  - less than or equal to 1,024, since there can be at most 1,024 threads issued at a time, per multiprocessor.

  If the above conditions are not satisfied, then there will be less than complete utilization of the hardware. Further, we need to ensure that the number of threads per block is at least 128, to allow enough instructions such that the scheduler can effectively overlap transfer and compute instructions. Finally, at most 8 blocks per multiprocessor can be active at a time.

- When the subroutine S is partitioned into smaller kernels, the data that is written by kernel k_1 and needs to be read by kernel k_2 will be stored in global memory. So we need to minimize the total amount of data transferred between kernels in this manner. Due to high global memory access latencies, this memory is accessed in a coalesced manner.
- To obtain maximal speedup, we need to ensure that the cumulative runtime over all kernels is as low as possible, after accounting for computation as well as communication.
- We need to ensure that the number of registers per thread is minimized such that the multiprocessors are not allotted less than 100% of the threads that they are configured to run with.
- Finally, we need to minimize the number of kernels K, since each kernel has an invocation cost associated with it. Minimizing K ensures that the aggregate invocation cost is low.

Note that the above guidelines often place conflicting constraints on the automatic kernel generation engine. Our kernel generation algorithm is guided by a cost function which quantifies these constraints and hence is able to obtain the optimal solution for the problem.

11.3.3 Automatic Kernel Generation Engine

The pseudocode for our automatic kernel generation engine is shown in Algorithm 13. The input to the algorithm is the subroutine S which needs to be partitioned into GPU kernels and the number N of independent calls of S that are made in parallel.

Algorithm 13 Automatic Kernel Generation(N, S)

```
BESTCOST ← ∞
G(V,E) ← extract_graph(S)
for K = K_min to K_max do
    P ← partition(G,K)
    Q ← make_acyclic(P)
    if cost(Q) < BESTCOST then
        golden_config ← Q
        BESTCOST ← cost(Q)
    end if
end for
generate_kernels(golden_config)
```

The first step of our algorithm constructs the companion control and dataflow graph $G(V,E)$ of the C program. This is done using the Oink [1] tool. Oink is a set of C++ static analysis tools. Each unique line l_i of the subroutine S corresponds to a unique vertex v_i of G. If there is a variable written in line l_1 of S which is read by line l_2 of S, then the directed edge $(v_1,v_2) \in E$. Each edge has a weight associated

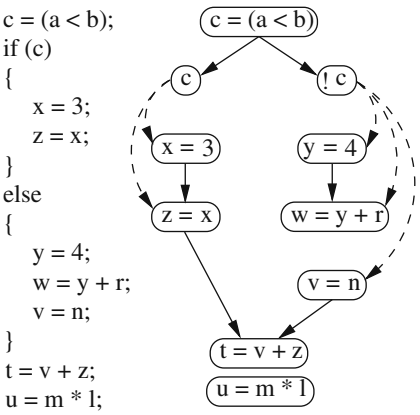


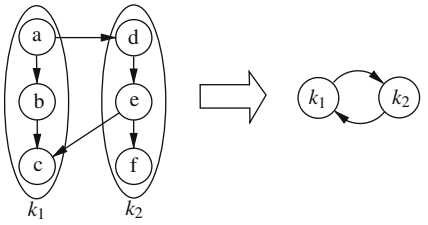
Fig. 11.1 CDFG example

with it, which is proportional to the number of bytes that are transferred between the source node and the sink node. An example code fragment and its graph G (with edge weights suppressed) are shown in Fig. 11.1.

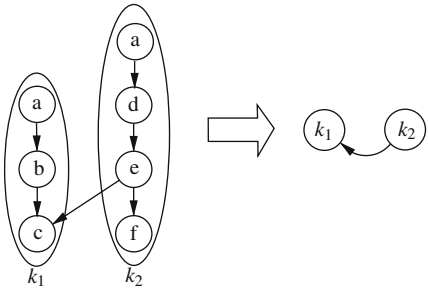
Note that if there are if–then–else statements in the code, then the resulting graph has edges between the node corresponding to the condition being checked and each of the statements in the then and else blocks, as shown in Fig. 11.1.

Now our algorithm computes a set P of partitions of the graph G, obtained by performing a K-way partitioning of G. We use hMetis [5] for this purpose. Since hMetis (and other graph-partitioning tools) operate on undirected graphs, there is a possibility of hMetis' solution being infeasible for our purpose. This is illustrated in Fig. 11.2. Consider a companion CDFG G which is partitioned into two partitions k_1 and k_2 as shown in Fig. 11.2a. Partition k_1 consists of nodes a, b, and c, while partition k_2 consists of nodes d, e, and f. From this partitioning solution, we induce a *kernel dependency graph (KDG)* $G_K(V_K,E_K)$ as shown in Fig. 11.2b. In this graph, $v_i \in V_K$ iff k_i is a partition of G. Also, there is a directed edge $(v_i,v_j) \in E_K$ iff $\exists n_p,n_q \in V$ s.t. $(n_p,n_q) \in E$ and $n_p \in k_i$, $n_q \in k_j$. Note that a cyclic kernel dependency graph in Fig. 11.2b is an infeasible solution for our purpose, since kernels need to be issued sequentially. To fix this situation, we selectively duplicate nodes in the CDFG, such that the modified KDG is acyclic. Figure 11.2c illustrates how duplicating node a ensures that the modified KDG that is induced (Fig. 11.2d) is acyclic. We discuss our duplication heuristic in Section 11.3.3.1.

In our kernel generation engine, we explore several K-way partitions. K is varied from $K_{\min}$ to a maximum value $K_{\max}$. For each of the explored partitions of the graph G, a cost is computed. This estimates the cost of implementing the partition on the GPU. The details of the cost function are described in Section 11.3.3.2. The lowest cost partitioning result *golden_config* is stored. Based on *golden_config*, we generate GPU kernels (using a PERL script). Suppose that *golden_config* was obtained by a k-way partitioning of S. Then each of the k partitions of *golden_config* yields a GPU kernel, which is automatically generated by our PERL script.



a) Partitioned CDFG b) Resulting KDG



c) CDFG after duplication d) Modified KDG

Fig. 11.2 KDG example

Data that is written by a kernel k_i and read by another kernel k_j ($k_i, k_j < k$) is stored in the GPU's global memory, in an array of length equal to the number of threads issued, and indexed at a location which is always aligned to a 32-byte boundary. This enables coalesced write and read accesses by threads executing kernel k_i and k_j, respectively. Since the cached memories are read-only memories, we cannot use them for communication between kernels. Also, since the given subroutine S is invoked N times on independent data, our generated kernels do not create any memory access conflicts when accessing global memory.

11.3.3.1 Node Duplication

To understand our node duplication heuristic, we first define a few terms. A *border node* (of a partition m of G) is a node $i \in V$ which has an outgoing edge to at least one node $j \in V$ such that j belongs to a partition $n \neq m$.

Our heuristic selectively duplicates border nodes until all cycles are removed. The duplication heuristic selects a node to duplicate based on the criteria below:

- a border node $i \in G$ (belonging to partition m, say) with an incoming edge from another partition n (which is a part of a cycle that includes m) and

- if the above criterion is not met, we look for border nodes i belonging to partitions which are on a cycle in the KDG, such that these nodes have a minimum number of incident edges $(z,i) \in E$, where $z \in G$ belongs to the same partition as i.

11.3.3.2 Cost of a Partitioning Solution

The cost of each partitioning solution is computed using several cost parameters, which are described next. In particular, our cost function C considers four parameters $\mathbf{x} = \{x_1,x_2,\ldots,x_4\}$. We consider a linear cost function, $C = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4$.

1. **Parameter** x_1: The first parameter of our cost function is the number of partitions being used. The GPU runtime is significantly modulated by this term, and hence it is included in our cost model.
2. **Parameter** x_2: This parameter measures the total time spent in communication to and from the device's global memory

$$x_2 = \left\lceil \frac{\sum_{i=1}^{K} (B_i)}{BW} \right\rceil$$

   Here B_i is the number of read or write transfers that are required for the partition i, and BW is the peak bandwidth for coalesced global memory transfers. Therefore the term x_2 represents the total amount of time that is spent in communicating data, when any one of the N calls of the subroutine S is executed.
3. **Parameter** x_3: The total computation time is estimated in this parameter. Note that due to node duplication, the total computation time is not a constant across different partitioning solutions. Let C_i be the number of GPU clock cycles taken by partition i. We estimate C_i based on the number of clock cycles for various instructions like integer and floating point addition, multiplication, and division, library functions for exponential and square root. This information is available from NVIDIA. Also let F be the frequency of operation of the GPU. Therefore, the time taken to execute the ith kernel is $\frac{C_i}{F}$. Based on this, $x_3 = \frac{\sum_{i=1}^{K} (C_i)}{F}$.
4. **Parameter** x_4: We also require that the average number of registers over all kernels is a small number. As discussed earlier, this is important to maximize speedup. This parameter (for each kernel) is provided by the *nvcc* compiler that is provided along with the CUDA distribution.

11.4 Experimental Results

Our kernel generation engine handles C programs. It handles non-recursive function calls (by value), pointers, structures, and if–else constructs. The kernel generation tool is implemented in Perl [10], and it uses hMetis [5] for partitioning and Oink [1] for generating the CDFG.

11.4.1 Evaluation Methodology

Our evaluation of our approach is performed in steps.

In the first step, we compute the weights α_1,$\alpha_2, \ldots, \alpha_4$. This is done by using a set L of benchmarks. For all these C-code examples, we generate the GPU code with 1, 2, 3, 4, . . ., 20 partitions (kernels). The code is then run on the GPU, and the values of runtime as well as all the **x** variables are recorded in each instance. From this data, we fit the cost function $C = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4$ in MATLAB. For any partitioning solution, we take the actual runtime on the GPU as the cost C, for curve-fitting. This yields the values of α_i.

In the second step, we use the values of α_i computed in the first step and run our kernel generation engine on a different set of benchmarks which are to be accelerated on the GPU. Again, we create 1, 2, 3, . . ., 20 partitions for each example. From these, we select the best three partitions (those which produce the three smallest values of the cost function). The kernel generation engine generates the GPU kernels for these partitions. We determine the best solution among the three (i.e., the solution which has the fastest GPU runtime) after executing them on the GPU.

Our experiments were conducted over a set of four benchmarks. These were as follows:

- **BSIM3**: This code computes the MOSFET model evaluations in SPICE [7]. The code computes three independent device quantities which are implemented in separate subroutines, namely BSIM3-1, BSIM3-2, and BSIM3-3.
- **MMI**: This code performs integer matrix–matrix multiplication. We experiment with MMI for matrices of various sizes (4×4 and 8×8).
- **MMF**: This code performs floating point matrix–matrix multiplication. We experiment with MMF for matrices of various sizes (4×4 and 8×8).
- **LU**: This code performs LU-decomposition, required during the solution of a linear system. We experiment with systems of varying sizes (matrices of size 4×4 and 8×8).

In the first step of the approach, we use the MMI, MMF, and LU benchmarks for matrices of size 4×4 and determined the values of α_i. The values of these parameters obtained were

$$\begin{aligned} \alpha_1 &= 0.6353, \\ \alpha_2 &= 0.0292, \\ \alpha_3 &= -0.0002, \text{ and} \\ \alpha_4 &= 0.1140. \end{aligned}$$

Now in the second step, we tested the usefulness of our approach on the remaining benchmarks (MMI, MMF, and LU for matrices of size 8×8, and BSIM3-1, BSIM3-2, and BSIM3-3 subroutines).

The results which demonstrate the fidelity of our kernel generation engine are shown in Table 11.1. In this table, the first column reports the number of partitions

Table 11.1 Validation of the automatic kernel generation approach

	MMI8		MMF8		LU8		BSIM3-1		BSIM3-2		BSIM3-3	
# Part.	Pred.	GPU time	Pred.	GPU time	Pred.	GPU time	Pred.	GPU time	Pred.	GPU time	Pred.	GPU time
1	√	**0.88**	√	**4.12**	√	**1.64**		41.40	√	**3.84**		53.10
2		0.96		**3.13**	√	**1.77**		**39.60**	√	4.25		**40.60**
3	√	**0.84**		4.25		2.76	√	43.70	√	4.34	√	43.40
4	√	**0.73**	√	6.04	√	6.12		44.10		**3.56**	√	**38.50**
5		1.53		7.42		**1.42**		43.70		**3.02**	√	42.20
6		1.14		5.06		8.53		43.40		4.33		43.50
7		1.53		6.05		5.69		43.50		4.36		43.70
8		1.04	√	**3.44**		7.65		45.10		11.32		98.00
9		1.04		8.25		5.13	√	**40.70**		4.61		49.90
10		1.04		15.63		10.00	√	**35.90**		24.12		57.50
11		1.04		9.79		14.68		43.40		35.82		43.50
12		2.01		12.14		16.18		44.60		40.18		**41.20**
13		1.14		13.14		13.79		43.70		17.27		44.00
14		1.55		14.26		10.75		43.90		52.12		84.90
15		1.81		11.98		19.57		45.80		36.27		53.30
16		2.17		12.15		20.89		43.10		4.28		101.10
17		2.19		17.06		19.51		44.20		18.14		46.40
18		1.95		13.14		20.57		46.70		34.24		61.30
19		2.89		14.98		19.74		49.30		35.40		46.80
20		2.89		14.00		19.15		52.70		38.11		51.80

being considered. Columns 2, 4, 6, 8, 10, and 12 indicate the three best partitioning solutions based on our cost model, for the MMI8, MMF8, LU8, BSIM3-1, BSIM3-2, and BSIM3-3 benchmarks, respectively. If our approach had perfect prediction fidelity, then these three partitioning solutions would have the lowest runtimes on the GPU. Columns 3, 5, 7, 9, 11, and 13 report the actual GPU runtimes for the MMI8, MMF8, LU8, BSIM3-1, BSIM3-2, and BSIM3-3 benchmarks, respectively. The three solutions that actually had the lowest GPU runtimes are highlighted in bold font in these columns.

Generating the partitioning solutions followed by automatic generation of GPU code (kernels) for each of these benchmarks was completed in less than 5 min on a 3.6 GHz Intel processor with 3 GB RAM and running Linux. The target GPU for our experiments was the NVIDIA Quadro 5800 GPU.

From these results, we can see the need for partitioning these subroutines. For instance in MMI8 benchmark, the fastest result obtained is with partitioning the code into 4 kernels, which makes it 17% faster compared to the runtime obtained using one monolithic kernel. Similar observations can be made for all other benchmarks. On average over these 6 benchmarks, our best predicted solution is 15% faster than the solution with no partitioning.

We can further observe that our kernel generation approach correctly predicts the best solution in three (out of six benchmarks), one of the best two solutions in five (out of six benchmarks), and one of the best three solutions in all six benchmarks. In comparison to the manual partitioning of BSIM3 subroutines, which was discussed

in Chapter 10, our automatic kernel generation approach obtained a partitioning solution that was 1.5× faster. This is a significant result, since the manual partitioning approach took us roughly a month for completion. In general, the GPU runtimes tend to be noisy, and hence it is hard to obtain 100% prediction fidelity.

11.5 Chapter Summary

GPUs are highly parallel SIMD engines, with high degrees of available hardware parallelism. These platforms have received significant interest for accelerating scientific software applications in recent times. The task of implementing a software application on a GPU currently requires significant manual intervention, iteration, and experimentation. This chapter presents an automated approach to partition a software application into kernels (which are executed in parallel) that can be run on the GPU. The input to our algorithm is a subroutine which needs to be accelerated on the GPU. Our approach automatically partitions this routine into GPU kernels. This is done by first extracting a graph which models the data and control dependencies in the subroutine in question. This graph is then partitioned. Any cycles in the graph induced by the partitions are removed by duplicating nodes. Various partitions are explored, and each is given a cost which accounts for GPU hardware and software constraints. Based on the least cost partition, our approach automatically generates the resulting GPU code. Experimental results demonstrate that our approach correctly and efficiently produces fast GPU code, with high quality. Our results show that with our partitioning approach, we can speed up certain routines by 15% on average when compared to a monolithic (unpartitioned) implementation. Our entire flow (from reading a C subroutine to generating the partitioned GPU code) is completely automated and has been verified for correctness.

References

1. Oink – A collaboration of C static analysis tools. `http://www.cubewano.org/oink`
2. Fisher, J.A., Ellis, J.R., Ruttenberg, J.C., Nicolau, A.: Parallel processing: A smart compiler and a dumb machine. SIGPLAN Notices **19**(6), 37–47 (1984)
3. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 215–226 (2004)
4. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A mapreduce framework on graphics processors. In: PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 260–269 (2008)
5. Karypis, G., Kumar, V.: A Software package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices. http://www-users.cs.umn.edu/∼karypis/metis (1998)
6. Kuck, Lawrie, D., Cytron, R., Sameh, A., Gajski, D.: The architecture and programming of the Cedar System. Cedar Document no. 21, University of Illinois at Urbana-Champaign (1983)
7. Nagel, L.: SPICE: A computer program to simulate computer circuits. In: University of California, Berkeley UCB/ERL Memo M520 (1995)

8. Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, Reading, MA (2005)
9. Sintorn, E., Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm. Journal of Parallel and Distributed Computing **68**(10), 1381–1388 (2008)
10. Wall, L., Schwartz, R.: Programming perl. O'Reilly and Associates, Inc., Sebastopol, CA (1992)

Chapter 12
Conclusions

In recent times, the gain in single-core performance of general-purpose microprocessors has declined due to the diminished rate of increase of operating frequencies. This is attributed to the power, memory, and ILP walls that are encountered as VLSI technology scales. At the same time, microprocessors are becoming increasingly complex with multiple cores being implemented on the same IC. This problem of reduced gains in performance in single-core processors is significant for EDA applications, since VLSI design complexity is continuously growing. In this monograph, we evaluated the viability of alternate platforms (such as custom ICs, FPGAs, and graphics processors) for accelerating EDA algorithms. We chose applications for which there is a strong motivation to accelerate, since they are used several times in the VLSI design flow, and have varied degrees of inherent parallelism in them. We studied two different categories of EDA algorithms:

- control dominated and
- control plus data parallel.

In particular, Boolean satisfiability (SAT), Monte Carlo based statistical static timing analysis, circuit simulation, fault simulation, and fault table generation were explored.

In Part I of this monograph, we discussed hardware platforms, namely custom-designed ICs, FPGAs, and graphics processors. These hardware platforms were compared in Chapter 2, using criteria such as their architecture, expected performance, programming model and environment, scalability, design turn-around time, security, and cost of hardware. In Chapter 3, we described the programming environment used for interfacing with the GPU devices.

In Part II of this monograph, three hardware implementations for accelerating SAT (a control-dominated EDA algorithm) were presented. A custom IC implementation of a hardware SAT solver was described in Chapter 4. This solver is also capable of extracting the minimum unsatisfiable core. The speed and capacity for our SAT solver obtained are dramatically higher than those reported for existing hardware SAT engines. The speedup was attributed to the fact that our engine performs the tasks of computing implications and determining conflicts in parallel, using a specially designed clause cell. Further, approaches to partition a SAT

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2_12,
© Springer Science+Business Media, LLC 2010

instance into banks and bin them into strips were developed, resulting in a very high utilization of clause cells. Also, through SPICE simulations we determined that the average power consumed per cycle by our SAT solver is under 1 mW, which further strengthens the practicality of our approach.

An FPGA-based approach for SAT was presented in Chapter 5. In this approach, the traversal of the implication graph as well as conflict clause generation is performed in hardware, in parallel. In our approach, clause literals are stored in FPGA slices. In order to solve large SAT instances, we heuristically partitioned the clauses into a number of bins, each of which could fit in the FPGA. This was done in a pre-processing step. The on-chip block RAM (BRAM) was used for storing all the bins of a partitioned CNF problem. The FPGA-based SAT solver implements a GRASP [6] like BCP engine, which performs non-chronological backtracks both within a bin and across bins. The embedded PowerPC processor on the FPGA performed the task of loading the appropriate bin from the BRAM, as requested by the hardware. Our entire flow was verified for correctness on a Virtex-II Pro based evaluation platform. We projected the runtimes obtained on this platform to an industry-strength XC4VFX140-based system and showed that a speedup of 17× can be obtained, over MiniSAT [1], a state-of-the-art software SAT solver. The projected system handles instances with as many as 280K clauses on 10K variables.

A SAT approach with a new GPU-enhanced variable ordering heuristic was presented in Chapter 6. Our approach was implemented in a CPU-based procedure which leverages the parallelism of a GPU. The CPU implements MiniSAT, a complete procedure, while the GPU implements SurveySAT, an approximate procedure. The SAT search is initiated on the CPU and after a user-specified fraction of decisions have been made, the GPU-based SurveySAT engine is invoked. Any new decisions made by the GPU-based engine are returned to MiniSAT, which now updates its variable ordering. This procedure is repeated until a solution is found. Our approach retains completeness (since it implements a complete procedure) but has the potential of high speedup (since the incomplete procedure is executed on a highly parallel graphics processor based platform). Experimental results demonstrate that on average, a 64% speedup was obtained over several benchmarks, when compared to MiniSAT.

In Part III of this monograph, several algorithms (with varying degrees of control and data parallelism) were accelerated using a graphics processor. Monte Carlo based SSTA was accelerated on a GPU in Chapter 7. In this approach we map Monte Carlo based SSTA to the large number of threads that can be computed in parallel on a GPU. Our approach performs multiple delay simulations of a single gate in parallel. It benefits from a parallel implementation of the Mersenne Twister pseudo-random number generator on the GPU, followed by Box–Muller transformations (also implemented on the GPU). We store the μ and σ of the pin-to-output delay distributions for all inputs and for every gate on fast cached memory on the GPU. In this way, we leverage the large memory bandwidth of the GPU. This approach was implemented on an NVIDIA GeForce GTX 280 GPU card and experimental results indicate that this approach can obtain an average speedup of about 818×

as compared to a serial CPU implementation. With the recently announced quad GTX 280 GPU cards, we estimate that our approach would attain a speedup of over 2,400×.

In Chapter 8, we accelerate fault simulation on a GPU. A large number of gate evaluations can be performed in parallel by employing a large number of threads on a GPU. We implemented a pattern- and fault-parallel fault simulator which fault-simulates a circuit in a forward levelized fashion. Fault injection is also performed along with gate evaluation, with each thread using a different fault injection mask. Since GPUs have an extremely large memory bandwidth, we implement each of our fault simulation threads (which execute in parallel with no data dependencies) using memory lookup. Our experiments indicate that our approach, implemented on a single NVIDIA GeForce GTX 280 GPU card, can simulate on average 47× faster when compared to an industrial fault simulator. On a Tesla (8-GPU) system [2], our approach can potentially be 300× faster.

The generation of a fault table is accelerated on a GPU in Chapter 9. We employ a pattern-parallel approach, which utilizes both bit parallelism and thread-level parallelism. Our implementation is a significantly modified version of FSIM [4], which is a pattern-parallel fault simulation approach for single-core processors. Our approach, like FSIM, utilizes critical path tracing and the dominator concept to prune unnecessary computations and thereby reduce runtime. We do not store the circuit (or any part of the circuit) on the GPU, and implement efficient parallel reduction operations to communicate data to the GPU. When compared to FSIM∗, which is FSIM modified to generate a fault table on a single-core processor, our approach (on a single NVIDIA Quadro FX 5800 GPU card) can generate a fault table (for 0.5 million test patterns) 15× faster on average. On a Tesla (8-GPU) system [2], our approach can potentially generate the same fault table 90× faster.

In Chapter 10, we study the speedup obtained when implementing the model evaluation portion of SPICE on a GPU. Our code is ported to a commercial fast SPICE [3] tool. Our experiments demonstrate that significant speedups (2.36× on average) can be obtained for the application. The asymptotic speedup that can be obtained is about 4×. We demonstrate that with circuits consisting of as few as about 1,000 transistors, speedups of about 3× can be obtained.

In Part IV of this monograph, we discussed automated acceleration of single-core software on a GPU. We presented an automated approach for GPU-based software acceleration of serial code in Chapter 11. The input to our algorithm is a subroutine which is executed multiple times, on different data, and needs to be accelerated on the GPU. Our approach aims at automatically partitioning this routine into GPU kernels. This is done by first extracting a graph, which models the data and control dependencies of the subroutine in question, and then partitioning it. Various partitions are explored, and each is assigned a cost which accounts for GPU hardware and software constraints, as well as the number of instances of the subroutine that are issued in parallel. From the least cost partition, our approach automatically generates the resulting GPU code. Experimental results demonstrate that our approach correctly and efficiently produces fast GPU code, with high quality. We show that with our partitioning approach, we can speed up certain routines by 15%

on average when compared to a monolithic (unpartitioned) implementation. Our entire technique (from reading a C subroutine to generating the partitioned GPU code) is completely automated and has been verified for correctness.

All the hardware platforms studied in this monograph require a communication link with a host processor. This link often limits the performance that can be obtained using hardware acceleration. The EDA applications presented in this monograph need to be carefully designed, in order to work around the communication cost and obtain a speedup on the target platform. Future-generation hardware architectures may have much lower communication costs. This would be possible, for example, if the host and the accelerator are to be implemented on the same die or share the same physical RAM. However, for the existing architectures, it is crucial to consider the cost of this communication while architecting any hardware-accelerated application.

Some of the upcoming architectures are the 'Larrabee' GPU from Intel and the 'Fermi' GPU from NVIDIA. These newer GPUs aim at being more general-purpose processors, in contrast to current GPUs. A key limiting factor of the current GPUs is that all the cores of these GPUs can only execute one kernel at a time. However, the upcoming architectures have a distributed instruction dispatch unit, allowing more than one kernel to be executed on the GPU at once (as shown conceptually in Fig. 12.1).

The block diagram of Intel's Larrabee GPU is shown in Fig. 12.2. This new architecture is a hybrid between a multi-core CPU and a GPU and has similarities to both. Like a CPU, it offers cache coherency and compatibility with the x86 architecture. However, it also has wide SIMD vector units and texture sampling hardware like the GPU. This new GPU has a 1,024-bit (512-bit each way) ring bus for communication between cores (16 or more) and to DRAM memory [5].

The block diagram of NVIDIA's Fermi GPU is shown in Fig. 12.3. In comparison to G80 and GT200 GPUs, Fermi has double the number of (32) cores per shared

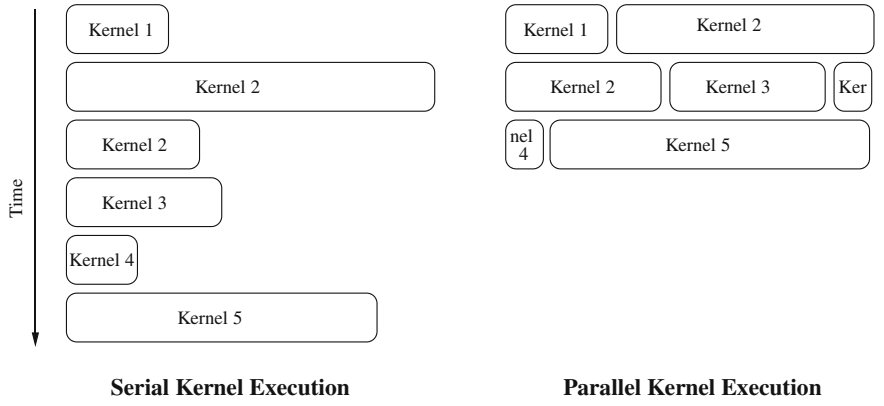


Fig. 12.1 New parallel kernel GPUs

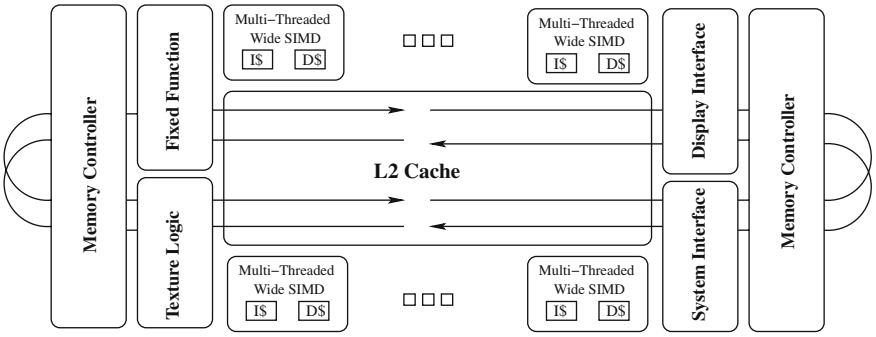


Fig. 12.2 Larrabee architecture from Intel

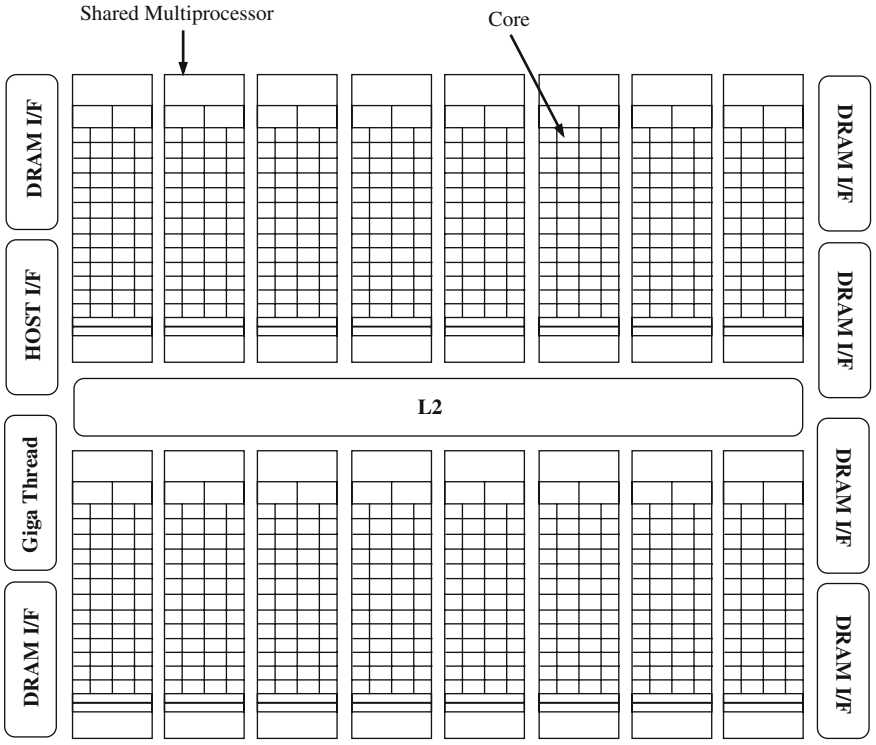


Fig. 12.3 Fermi architecture from NVIDIA

multiprocessor (SM). The block diagram of a single SM is shown in Fig. 12.4 and the block diagram of a core within an SM is shown in Fig. 12.5.

With these upcoming architectures, newer approaches for hardware acceleration of algorithms would become viable. These approaches could exploit the more general computing paradigm offered by the newer architectures. For example, the close coupling between the GPU and the CPU (which reside on the same die) would

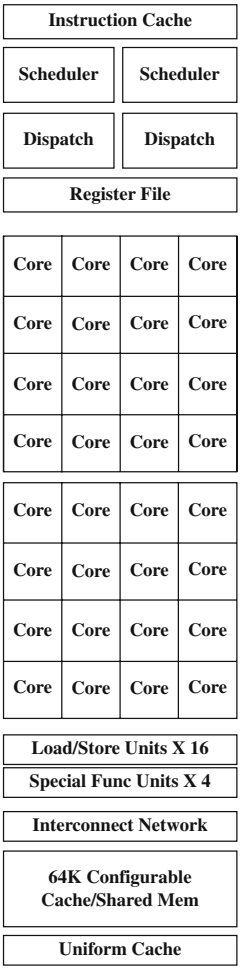


Fig. 12.4 Block diagram of a single shared multiprocessor (SM) in Fermi

reduce the communication cost. Also, in these upcoming architectures the instruction dispatch unit is distributed, and the instruction set is more general purpose. These enhancements would enable a more general computing paradigm (in comparison to the SIMD paradigm for current GPUs), which in turn would enable acceleration opportunities for more EDA applications.

The approaches presented in this monograph collectively aim to contribute toward enabling the CAD community to accelerate EDA algorithms on modern hardware platforms. Our work demonstrates techniques to rearchitect several EDA algorithms to maximally harness their performance on the alternative platforms under consideration.

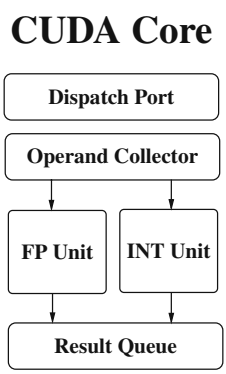


Fig. 12.5 Block diagram of a single processor (core) in SM

References

1. http://www.cs.chalmers.se/cs/research/formalmethods/minisat/main.html. The MiniSAT Page
2. NVIDIA Tesla GPU Computing Processor. `http://www.nvidia.com/object/IO_43499.html`
3. OmegaSim Mixed-Signal Fast-SPICE Simulator. `http://www.nascentric.com/product.html`
4. Lee, H.K., Ha, D.S.: An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation. In: Proceedings of the IEEE International Test Conference on Test, pp. 946–955. IEEE Computer Society, Washington, DC (1991)
5. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A many-core x86 architecture for visual computing. ACM Transactions on Graphics **27**(3), 1–15 (2008)
6. Silva, M., Sakallah, J.: GRASP-a new search algorithm for satisfiability. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 220–7 (1996)

Index

A
Accelerators, 9
ACML-GPU, 15
Activity, 93
Algorithm parallel, 120, 121, 134
Amdahl's Law, 158, 170
Application specific, 64
Arrival time, 110
Assignment, 31, 37, 40

B
Backtracking, 32
Bandwidth, 13
Bandwidth minimization, 52
Bank conflict, 27
BCP, 32, 37, 40
Bias
  survey propagation, 89
Bins, 64
Bin packing, 52, 70
Bin utilization, 74
Bit parallel, 135, 146
Block, 28
Block-based
  SSTA, 108
Board test, 15
Boolean Constant Propagation, *see* BCP
Boolean Satisfiability, *see* SAT
Box-Muller, 101
BRAM, 11, 14, 32, 63, 66, 72, 78
Brook+, 15
BSIM3
  SPICE, 158
BSIM4
  SPICE, 158
Bulldog Fortran, 171

C
Capacity, 31, 35
CDFG, 160
Clause, 31
Clock speed, 11
CNF, 31, 34
Co-processors, 9
Compilers, 16
Complete
  SAT, 83, 85
Conflict, 37, 40, 42, 44, 71
Conflict clause, 31
Conflict clause generation, 33, 64
Conjunctive Normal Form, 34
Constant Memory, 26, 161
Control and dataflow graph, 173
Control dominated
  EDA, 3
Control plus data parallel
  EDA, 3
Core, 185
Critical line
  critical path tracing, 138
Critical path tracing, 138
CUBLAS, 15
CUDA, 15, 24
CUFFT, 15
Cumulative detectability, 138
Custom IC, 7, 10, 33

D
Data parallel, 28, 106, 120, 122, 134
Debuggers, 16
Decision engine, 37, 39, 49, 70
Decision level, 39, 67
Decisions
  SAT, 32
Detectability, 138
DFF, 11
DIMACS, 45
Dimblock, 29

K. Gulati, S.P. Khatri, *Hardware Acceleration of EDA Algorithms*,
DOI 10.1007/978-1-4419-0944-2,

© Springer Science+Business Media, LLC 2010

Dimensionality, 29
Dimgrid, 29
Divide, 12
Dominator, 138
DPLL, 85
DRAM, 14, 66, 184
Dropped
    fault table, 134
Dynamic
    power, 10
Dynamic bulk modulation, 10

E
EDA, 3
Embedded processor, 10

F
Factor Graph, 87
Fault detection, 134
Fault diagnosis, 134
Fault dropping, 134
Fault grading, 102, 120
Fault injection, 135
Fault parallel
    data parallel, 120
Fault simulation, 4, 119
Fault table, 4, 134
Fermi, 184
Fingerprinting, 19
FPGA, 3, 7, 10, 32
Function
    Factor Graph, 87

G
Global Memory, 13, 27, 110, 159
GPGPU, 3
Graphics Processors, *see* GPU
GRASP, 35, 64, 85
Grid, *see* dimgrid
GridSAT, 87
GSAT, 85

H
Hardware
    IP cores, 15
HDL, 10, 14, 19
Hybrid
    SAT, 85

I
Immediate dominator
    dominator, 138
Implication, 37, 40, 44
Implication graph, 31, 33, 37, 50, 64
Infringement
    security, 19
Input vector control, 10
Instance specific, 64
Inter-bin
    non-chronological, 32
Intra-bin
    non-chronological, 32
IP cores, 15

K
Kernel, 28, 167, 184

L
Larrabee, 184
Latency, 11, 13
Leakage
    power, 10
Levelize, 112
Literal, 37
    free literal, 41
Local memory, 12, 27
Logic analyzers, 15
Lookup table, 11, 106, 120
LUT, 12

M
Memory bandwidth, 1, 13
Memory wall, 1
Mersenne Twister, 101, 106, 112
MIMD, 171
Minimum unsatisfiable core, 31, 33, 53
MiniSAT, 85
MNA
    SPICE, 154
Model evaluation, 154
Model parallel, 122, 134
Monte Carlo, 4
    SSTA, 101, 106
Moore's Law, 24
MOPs, 17
MOPs per watt
    MOPs, 17
Multi-GPU, 16
Multi-port
    memory, 20
Multiprocessor, 12, 24
MUX, 11

N
Newton-Raphson, 154
NMOS
    passgates, 11

Non-chronological backtrack, 32, 43, 45, 64, 68, 85
Non-recurring engineering, 10, 18
Non-volatile
memory, 20

O
Off-chip, 14
On-chip, 14
OPB, 67, 72

P
Paging, 12
Parafrase, 170
Parallel
SAT, 85
Partition, 32, 35, 63, 78
Pass/fail fault dictionary, 134
Path-based
SSTA, 108
Pattern parallel
data parallel, 120
PCI, 15
PCI-X
PCI, 15
Pipeline, 11
Piracy
security, 19
PLB, 67, 72
PLB2OPB bridge, 72
Power, 10, 56
average power, 58
Power delay product, 18
Power gating, 10
Power wall, 1
PowerPC, 32
Precharged, 39
Predischarged, 39
Process variations, 106
Processor, 24
Profiling
code, 16
Programmable, 12
Prototyping, 16

Q
QuickPath Interconnect, 18

R
Random
variations, 106
Re-programmability, 19
Reconfigurable logic
FPGA, 11
Reconfigure, 12
Reduced OR, 144
Register, 26, 172
Resolution, 36
Reuse-based design, 19

S
Sample parallelism, 106
SAT, 4, 31, 33, 34, 36
3-SAT, 36
Scalability, 15, 31, 35, 66
Scattered reads, 29
SEE, 18, 114
Self-test, 15
Sensitive input, 138
Shared Memory, 26, 27, 110
Shared multiprocessor, 185
SIMD, 3, 18, 29
Software
IP cores, 15
Span, 69
Speedup, 31
SPICE, 31, 153
Square root, 12
SRAM, 11
SSTA, 4, 101, 106
STA, 101, 106
SPICE, 154
Stem, 137
Stem region, 138, 143
Stochastic
SAT, 83, 85
Subroutine, 167
Subsumption
resolution, 56
Successive chord, 156
Supply voltage, 10
Survey propagation, 84
Surveys
survey propagation, 88
Synchronization points, 29
Synchronize, 28
System test, 15
Systematic
variations, 106

T
Termination cell, 40
Texture fetching
Texture Memory, 27
Texture Memory, 26, 110, 155, 160
Thread, 28, 146
Thread block, 28
Thread parallel, 135

Thread scheduler, 29
Throughput, 11
Time slicing, 29
Tree Factor Graph, 87

U
Unate covering, 134

V
Variable, 31, 37
    Factor Graph, 87
Variable ordering
    SAT, 32
Variable Vt, 10
Variations, 106
Virtual memory, 12
VLIW, 171
VLSI, 106
VSIDS, 93

W
WalkSAT, 85, 90, 96
Warp size, 29
Warps, 29
Watermarking, 19

X
XC2VP30
    FPGA, 32